ZPL

- It’s like programming languages you know
  - Imperative statements, arithmetic/logical expressions...
  - Declarations … typed about as strongly as C
  - The usual control structures, procedures, I/O, ...
  - A syntax people complain about. Of course!

- It’s like nothing you’ve ever programmed...
  - Many new features... regions, flooding, remap, etc.

ZPL’s Goals: Run fast (performance) everywhere (portability) with minimal programming effort (convenience)

ZPL ...

- Is an array language -- whole arrays are manipulated with primitive operations
- Requires new thinking strategies --
  - Forget one-operation-at-a-time scalar programming
  - Think of the computation globally -- make the global logic work efficiently and leave the details to the compiler
- Is parallel, but there are no parallel constructs in the language; the compiler...
  - Finds all concurrency
  - Performs all interprocessor communication
  - Implements all necessary synchronization (almost none)
  - Performs extensive parallel and scalar optimizations

ZPL Basics ...

ZPL has the usual stuff

- Datatypes: **`boolean`**, **`float`**, **`double`**, **`quad`**, **`complex`**, signed and unsigned integers: **`sbyte`**, **`ubyte`**, **`integer`**, **`uinteger`**, **`char`**, …
- Operators:
  - Unary: **`+, -, !`**
  - Binary: **`+, -, *, /, ^, %, &, |`**
  - Relational: **`<, <=, =, !=, >=, >=`**
  - Bit Operations: **`bnot(), band(), bor(), bxor(), bsl(), bsr()`**
  - Assignments: **`:=, +=, -=, *=, /=, %=, &=, |=`**
- Control Structures: if-then-[elsif]-else, repeat-until, while-do, for-do, exit, return, continue, halt, begin-end

ZPL Basics (continued)

- White space is ignored
- All statements are terminated by semicolon (;)
- Comments are

  `--` to the end of the line

  `/* */` all text within pairs including newlines
- All variables must be declared using **`var`**
- Names are case sensitive
- Programs begin with `program <name>;`

  the procedure with <name> is the entry point
- Statements execute sequentially

To Guide The Compiler ...

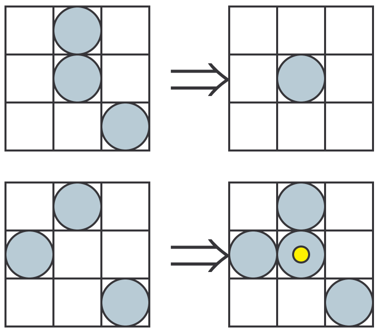
ZPL provides high level mechanisms to express computation with a minimum of serialization

- New concepts are needed
  - – Regions
  - – Directions
  - – Global and partial reductions
  - – Many others
- Best introduced by example …
  - Conway's Game of Life

1) Survive with 2 or 3 neighbors

2) Birth with exactly 3 neighbors

Goal: Focus on "what," not "how"



A Global Solution

- How to represent the world (**`TW`**): Array of bits, 1=organism, 0=empty; toroidal
- Decisions must be based on how many neighbors each position has, so must compute neighbor count (**`NN`**) for whole array
- Given array of neighbor counts, apply the rules to create next generation
- Repeat until no organisms remain--0 array

Expressing the Global Rules Globally

Conway's Life: The World is bits

```
[R] repeat
      NN := TW@^NW + TW@^N  + TW@^NE
           + TW@^W    +            TW@^E
           + TW@^SW  + TW@^S  + TW@^SE;
      TW := (TW & NN = 2) | (NN = 3);
    until ! (|<< TW);
```

Add up neighbor bits

Apply rules to live by

"Or" bits in world to see if any alive

Expressing the Global Rules Globally

Conway's Life: The World is bits

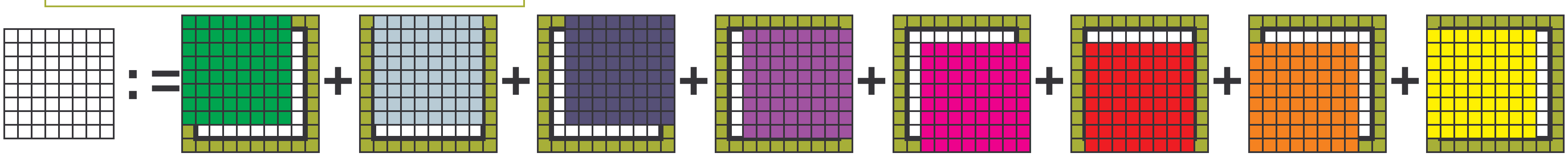
```
[R] repeat
        NN := TW@^NW + TW@^N  + TW@^NE
              + TW@^W    +             TW@^E
              + TW@^SW  + TW@^S  + TW@^SE;
        TW := (TW & NN = 2) | (NN = 3);
      until ! (|<< TW);
```

Add up neighbor bits

Apply rules to live by

"Or" bits in world to see if any alive

Edges wrap around ⇓



Cartoon of counting neighbors: Array of NW neighbors+ array of north neighbors+array of NE neighbors+...

Game of Life … the Program

Declarations are key to setting up an effective context

```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];

direction NW = [-1,-1]; N = [-1, 0]; NE = [-1, 1];
           W = [ 0,-1];               E = [ 0, 1];
          SW = [ 1,-1]; S = [ 1, 0]; SE = [ 1, 1];

var   NN : [R] ubyte; TW : [R] boolean;

procedure Life();
      [R] begin
          /* Read in the data */

           repeat
             NN :=  TW@^NW + TW@^N  + TW@^NE
                  + TW@^W           + TW@^E
                  + TW@^SW + TW@^S  + TW@^SE;
             TW := (NN=2 & TW) | (NN=3);
           until ! |<<TW;

          end;
```

Declaration Basics

- config: define default vals but revise on command line
- region … define index set it's like an array w/o data
- direction … define vector pointing in *index* space

```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];

direction NW = [-1,-1]; N = …
           W = [ 0,-1];
          SW = [ 1,-1]; S = …

var    NN : [R] ubyte; TW : …

procedure Life();
     [R] begin
          /* Read in the data
```

- regions used for two purposes … declarations and controlling computation

Regions, A Key ZPL Idea

- Regions are index sets
- Any number of dimensions, any bounds
  - region V = [1..n];
  - region R = [1..m, 1..m]; BigR = [0..m+1,0..m+1];
  - region Left = [1..m, 1];
  - region Odds = [1..n by 2];
- Short names are preferred--regions are used everywhere--and capitalization is a coding convention
- Naming regions is recommended but literals are OK

Using Regions to Declare Arrays

- Regions are used to declare arrays … it's like adding data to the indices
- Capitals are used by convention to separate arrays from scalars
- Named or literal regions are OK

```
var A, B, C : [R] double;
var Seq : [V] boolean;
var Huge : [0..2^n, -5..5] float;
```

- Regions are used once; no array has more than one region component
- Regions are a source of parallelism…

Regions Control Computation

- Statements containing arrays need a region to specify which items participate

```
[1..n,1..n] A := B + C;
        [R] A := B + C;        -- Same as above
```

- Regions are scoped

```
[R] begin
        …
[Left]  …
      end;
```

All array computations in compound statements are performed over indices in [R], except statement prefixed by [Left]

- Operations over region elements performed in parallel

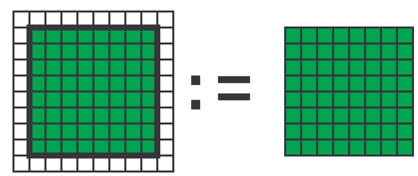
Parallelism In Statement Evaluation

- Let A, B be arrays over [1..n,1..n], and C be an array over [2..n-1,2..n-1] as in

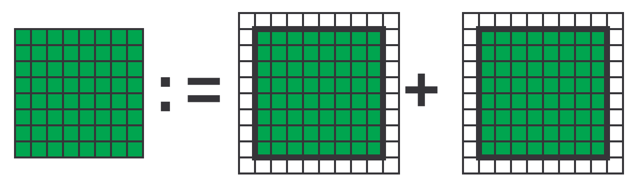
  var A, B : [1..n,1..n] float; C : [2..n-1,2..n-1] float;

- Then

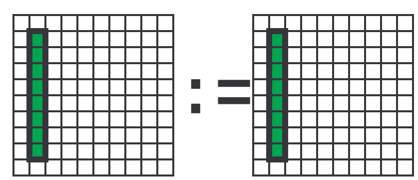
  [2..n-1,2..n-1] A := C;



  [2..n-1,2..n-1] C := A + B;



  [2..n-1, 2] A := B;

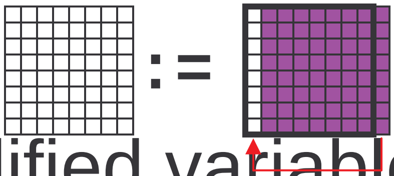


@ Uses Regions & Directions

The @ operator combines regions with directions to allow references to neighbors

- Two forms, standard(@) and wrapping(@^)
- Syntax: A@east   A@^east
- Semantics: the direction is added to elements of region giving new region, whose elements are referenced; think of a region translation

  [1..n,1..n] A := A@^east; -- shift array left with wrap around



- @-modified variables can appear on l or r of :=

Parallelism In Statement Evaluation

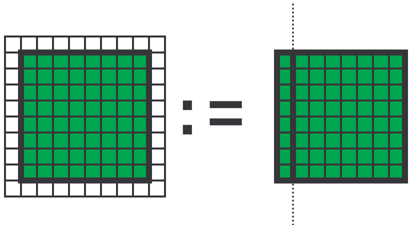
- Let

  var A, B : [1..n,1..n] float; C : [2..n-1,2..n-1] float;

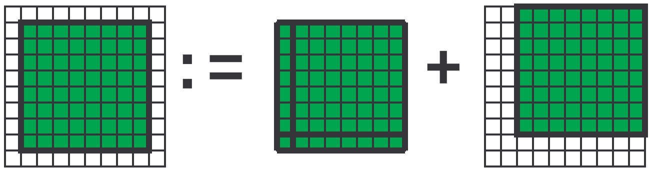
  direction east = [0,1]; ne = [-1,1];

- Then

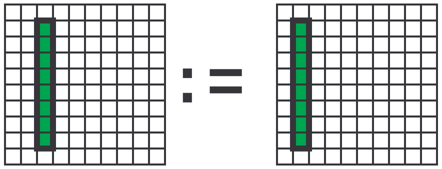
  [2..n-1,2..n-1] A := C@^east;



  [2..n-1,2..n-1] A := C@^ne + B@^ne;



  [2, 2..n-1] A@east := B;



Reductions, Global Combining Operations

- Reduction (<<) “reduces” the size of an array by combining its elements
- Associative (and commutative) operations are +<<, *<<, &<<, |<<, max<<, min<<

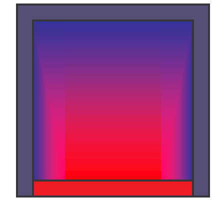
```
[1..n, 1..n] biggest := max<<A;
[R]          all_false := |<< TW;
```

- All elements participate; order of evaluation is unspecified … caution floating point users
- ZPL also has partial reductions, scans, partial scans, and user defined reductions and scans

Socrates: Unexamined Life Not Worth...

```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];

direction NW = [-1,-1]; N = [-1, 0]; NE = [-1, 1];
           W = [ 0,-1];               E = [ 0, 1];
          SW = [ 1,-1]; S = [ 1, 0]; SE = [ 1, 1];

var   NN : [R] ubyte; TW : [R] boolean;

procedure Life();
      [R] begin
          /* Read in the data */

           repeat
             NN :=  TW@^NW + TW@^N  + TW@^NE
                  + TW@^W           + TW@^E
                  + TW@^SW + TW@^S  + TW@^SE;
             TW := (NN=2 & TW) | (NN=3);
           until ! |<<TW;

          end;
```

Applying Ideas: Jacobi Iteration

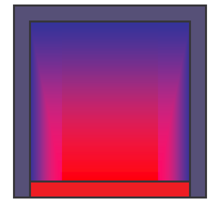


- Model heat defusing through a plate
- Represent as array of floating point numbers
- Use a 4-point stencil to model defusing
- Main steps when thinking globally

Initialize
Compute new averages
Find the largest error
Update array
… until convergence

The “High Level” Logic Of J-Iteration

```
program Jacobi;
config var n : integer = 512;
         eps : float = 0.00001;

region     R = [1..n, 1..n];
        BigR = [0..n+1,0..n+1];
direction  N = [-1, 0];  S = [ 1, 0];
           E = [ 0, 1];  W = [ 0,-1];
var     Temp : [R] float;
           A : [BigR] float;
         err : float;

procedure Jacobi();
     [R] begin
  [BigR] A := 0.0;
[S of R] A := 1.0;
          repeat
            Temp := (A@N + A@E + A@S + A@W)/4.0;
            err  := max<< abs(Temp - A);
            A    := Temp;
          until err < eps;
        end;
end;
```



Initialize
Compute new averages
Find the largest error
Update array
… until convergence

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 across 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

Index1 ...

- ZPL comes with "constant arrays" of any size
- Index*i* means indices of the i[th] 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

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	6	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

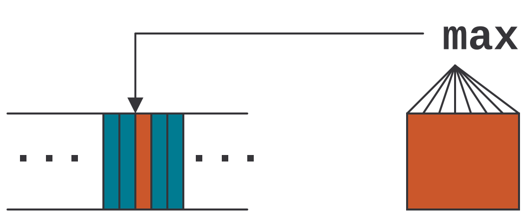
The purpose of keeping MaxC a 2D array is control how it is allocated

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

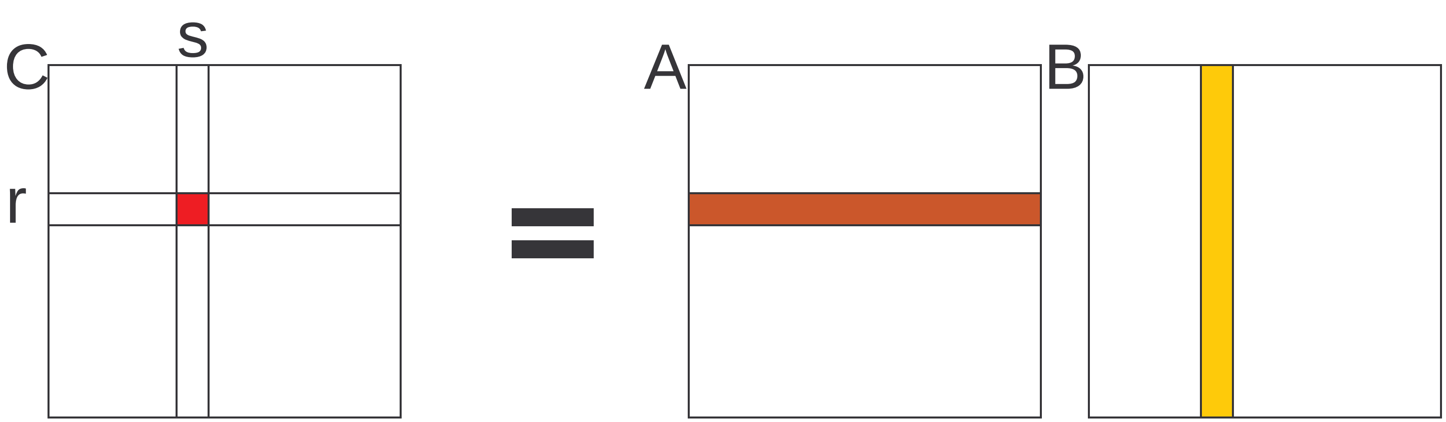
- Scaling a value by max of row w/o flooding:

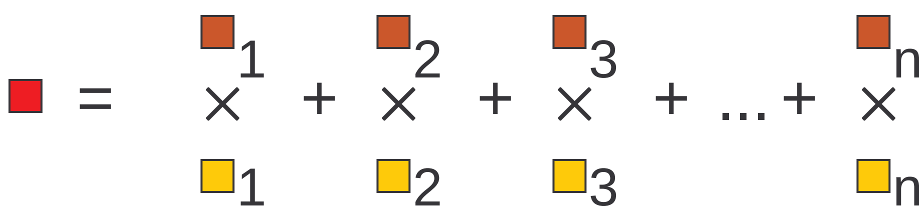
```
   [1..m,*]   MaxC := max<< [1..m,1..n] A;
[1..m,1..n]      A := A / MaxC;     --Scale A;
```

The promotion of flooded arrays is only logical

Recall Matrix Multiplication (MM)

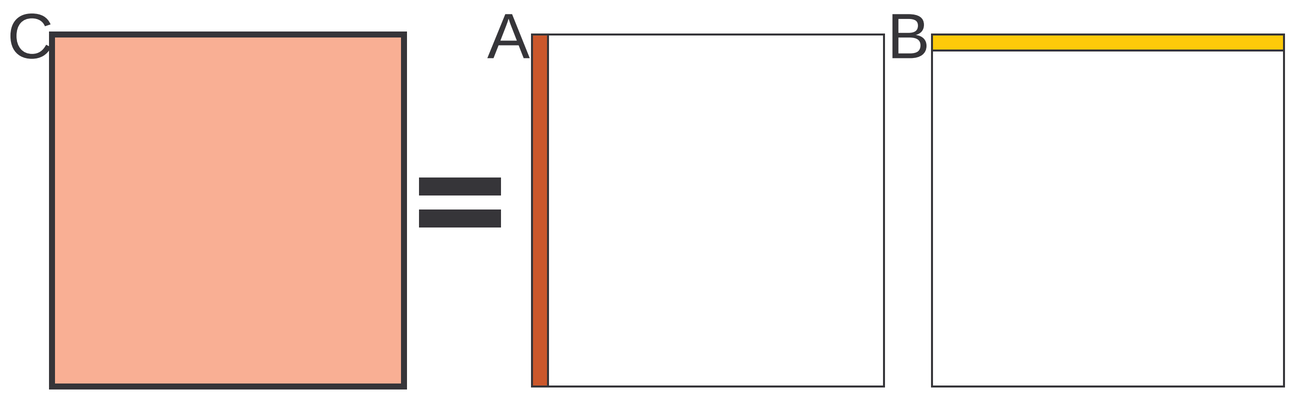
- For $n \times n$ arrays A and B, compute C = AB where $c_{rs} = \sum_{1 \le k \le n} a_{rk} b_{ks}$





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;
```

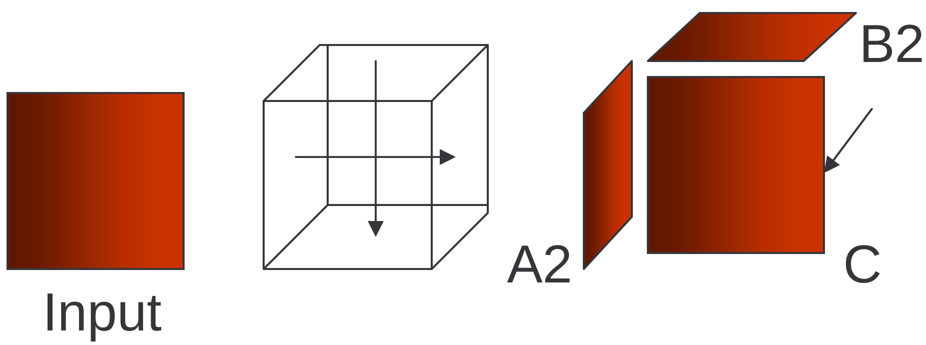
--- or, more simply ---

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

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);
```

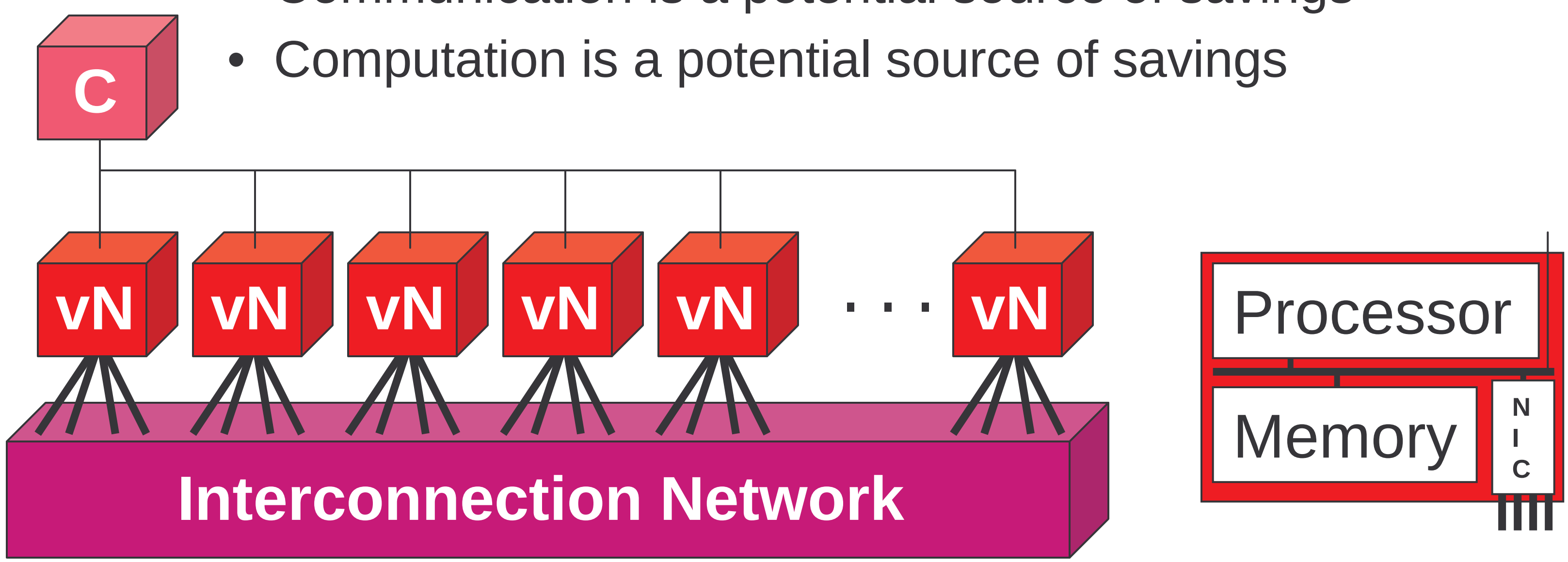


Optimizations of ZPL

- C, Java and most sequential languages operate on one scalar value at a time
  - Compilation focuses on single operations
  - Optimization has limited impact … combine two ops or remove an op or load saves one instruction
  - It's hard to see the forest for the trees
- ZPL and other array languages specify computation in large units … optimizations can have a large impact

Two Types of Costs

- Parallel computation differs from sequential computation in that interprocessor communication is *pure* overhead …
- For parallel languages
  - Communication is a potential source of savings
  - Computation is a potential source of savings



Looking Closer at Costs

Consequences of two forms of improvement

- Removing communication is always a win
- Because of multiple processors it's possible to replace comm with comp is usually a win
  - Sequential computation like a loop i := i + 1
- Moving communication can improve performance
  - Comm is performed by co-processor via DMA so processor can continue to work
  - Prefetching and pipelining can help

All scalar optimizations still benefit

Bumpers and Walkers

Recall "loop induction variable elimination" removed explicit index references, replacing them with pointer … ZPL applies this a lot

```
[prev of R] begin
               SampleT    := 0.0;
               SampleXPos := 0.0;
               SampleYPos := 0.0;
            end;
```

```
for (i=p.o.R.mylo;i<p.o.R.myhi;i++){
    SampleT[i]=0.0;}
for (i=p.o.R.mylo;i<p.o.R.myhi;i++){
    SampleXPos[i]=0.0;}
for (i=p.o.R.mylo;i<p.o.R.myhi;i++){
    SampleYPos[i]=0.0;}
```

Loop Fusion

Classic: consecutive loops over the same range can be merged, giving a longer loop body with (hopefully) more straight line code

```
for (i=p.o.R.mylo;i<p.o.R.myhi;i++){
    SampleT[i]=0.0;}
for (i=p.o.R.mylo;i<p.o.R.myhi;i++){
    SampleXPos[i]=0.0;}
for (i=p.o.R.mylo;i<p.o.R.myhi;i++){
    SampleYPos[i]=0.0;}
```

```
for (i=p.o.R.mylo;i<p.o.R.myhi;i++){
    SampleT[i]=0.0;
    SampleXPos[i]=0.0;
    SampleYPos[i]=0.0;}
```

Array Contraction

- Classic: Reduce an array (temp) to a scalar to improve locality and put variable in register

```
[R] T1 := (A + A@east)/2;
    T2 := (A + A@west)/2;
     A := max(T1,T2);
```

```
for (i=R.mylo;i<R.myhi;i++){
    T1[i]=((A[i]+A[i+1])/2;}
for (i=R.mylo;i<R.myhi;i++){
    T2[i]=((A[i]+A[i-1])/2;}
for (i=R.mylo;i<R.myhi;i++){
    A[i]=max(T1[i],T2[i]);}
```

- First, fuse the loops

Array Contraction, continued

- Fused loops:

```
for (i=R.mylo;i<R.myhi;i++){
    T1[i]=((A[i]+A[i+1])/2;
    T2[i]=((A[i]+A[i-1])/2;
    A[i]= max(T1[i],T2[i]);}
```

- Discover that T1, T2 not live after loop
- Analyze references … what values are needed to compute A[i]? A[i], A[i-1], A[i+1]
- Create code to save values

Array Contraction, continued

… And reduce T1 and T2 to scalars t1 and t2

```
    ai_west = A[R.mylo-1];
    ai      = A[R.mylo];
for (i=R.mylo;i<R.myhi;i++){
    ai_east = A[i+1];
    t1      =((ai+ai_east)/2;
    t2      =((ai+ai_west)/2;
    A[i]    = max(t1,t2);
    ai_west = ai;
    ai      = ai_east;
}
```

Compiler Created Temps

- Suppose that rather than writing

```
[R] T1 := (A + A@east)/2;
    T2 := (A + A@west)/2;
     A := max(T1,T2);
```

- The programmer had written

```
[R] A := max(A + A@east, A + A@west)/2;
```

- The compiler would have generated a (single) array temporary since A is on the left and right

Factor-Join Optimizations

- Consider a bounding box ZPL computation

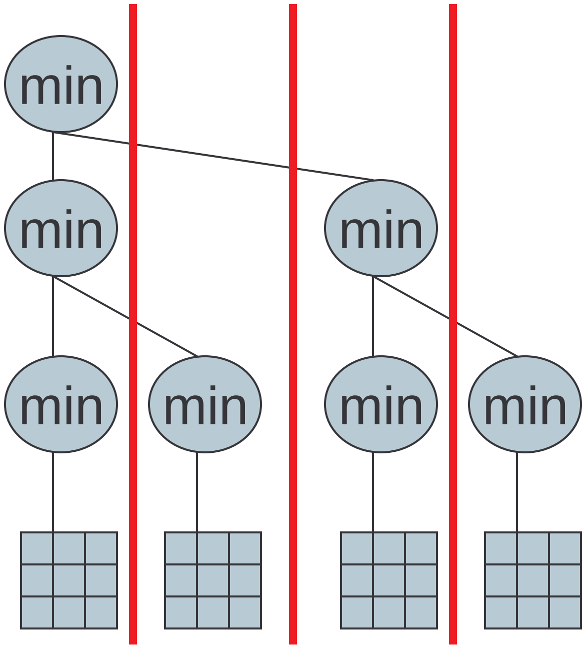
```
type point = record
   x : float;
   y : float;
end;  …
lox := min<<Pts.x;
loy := min<<Pts.y;
hix := max<<Pts.x;
hiy := max<<Pts.y;
```

var Pts : [R] point;

Factor-Join Optimizations

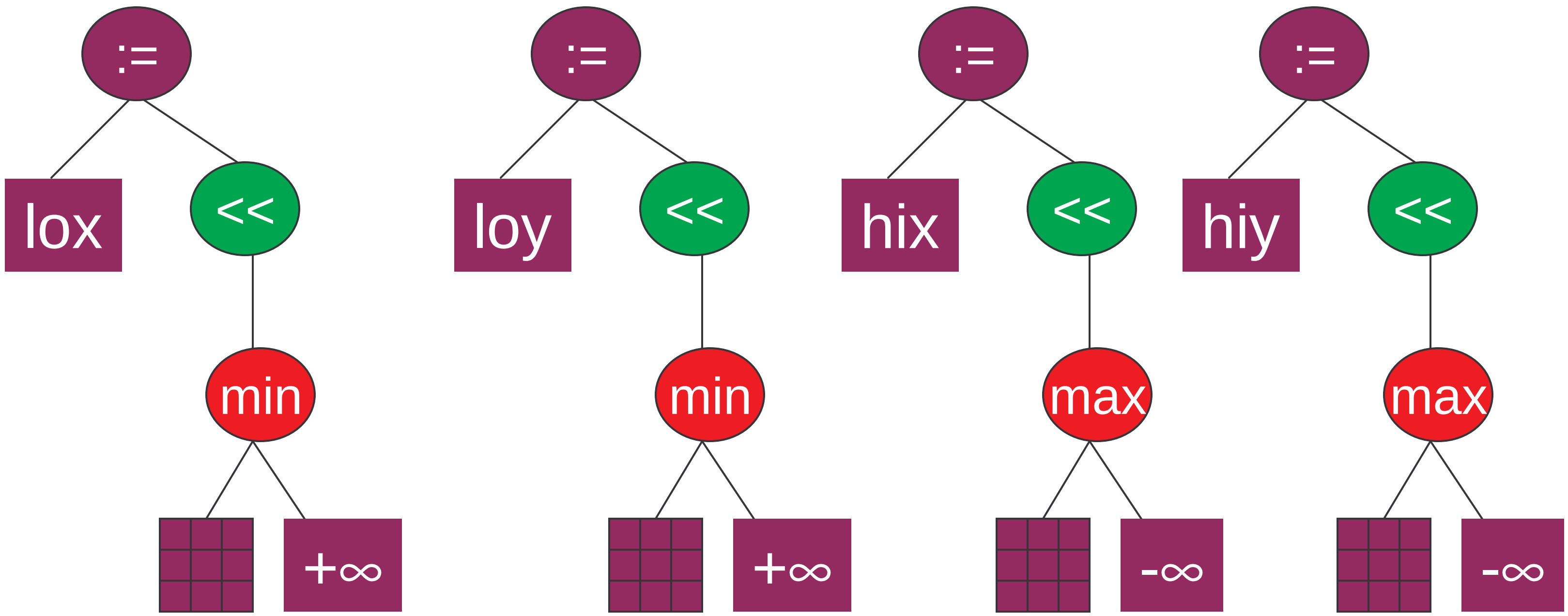
- Consider a bounding box ZPL computation

```
type point = record
   x : float;
   y : float;
end;  …
lox := min<<Pts.x;
loy := min<<Pts.y;
hix := max<<Pts.x;
hiy := max<<Pts.y;
```



IR for Macro Operations

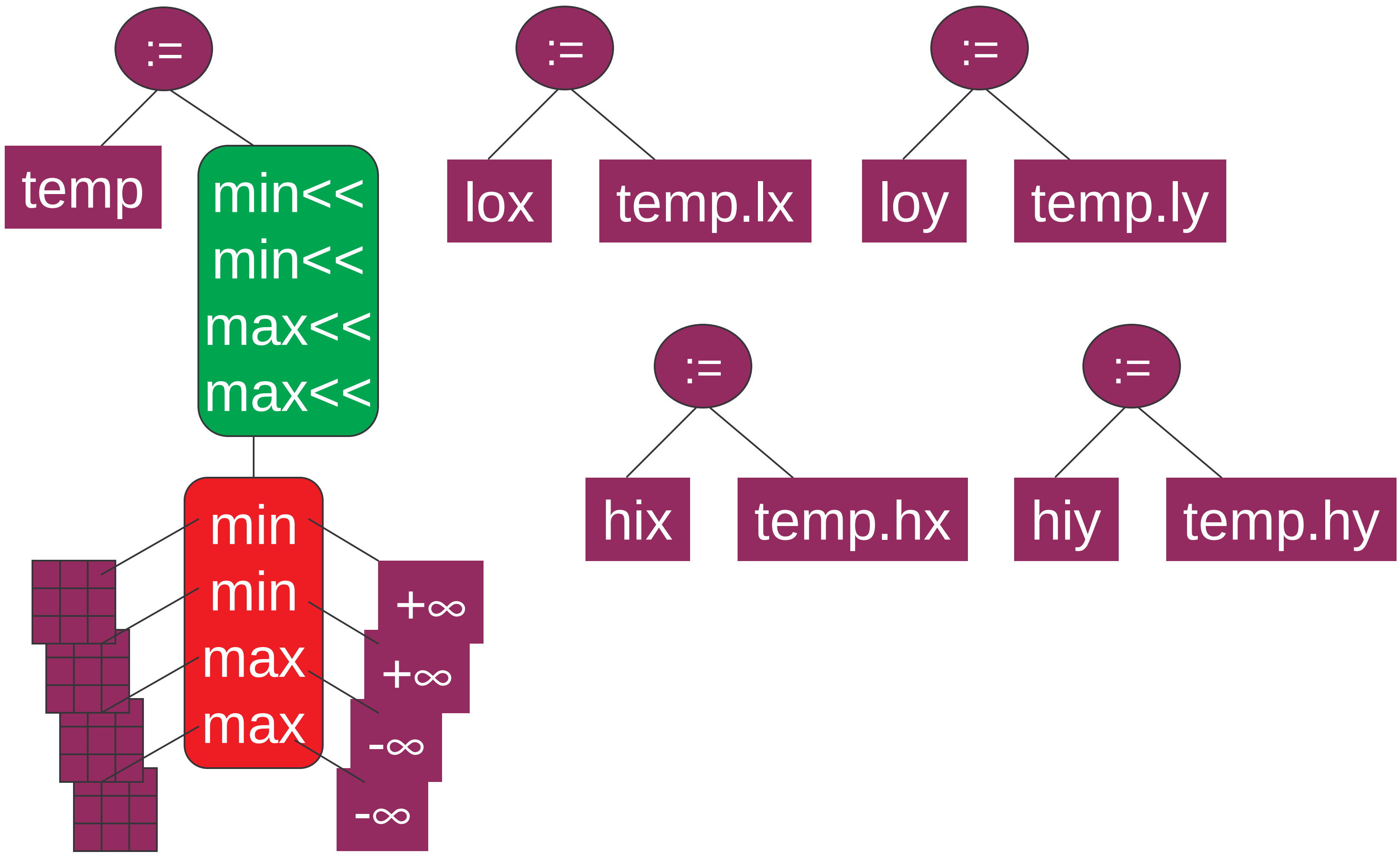
- Express the operations at large grain



Factor Join

- Recognize that communication and array traversals are expensive operations that can benefit from combining
  - Reductions/Scans can be merged because data size is usually small relative to packet capacity
  - Merging array traversals improves cache performance
  - Etc.
- Factor array operations into components, and join into new “merged” operations

IR for Macro Operations



Recall Conway's Life Program...

Conway's Life: The World is bits

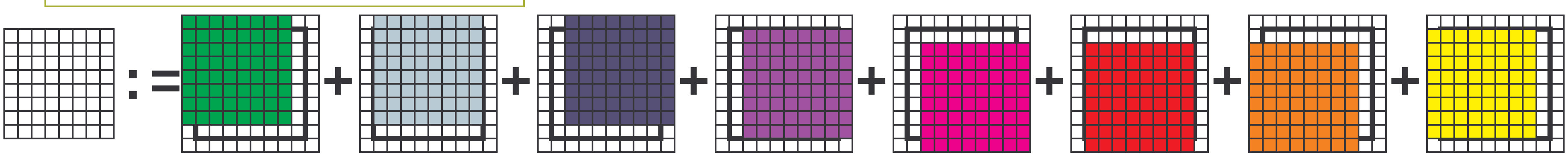
Add up neighbor bits

```
[R] repeat
      NN := TW@^NW + TW@^N  + TW@^NE
            + TW@^W    +            TW@^E
            + TW@^SW  + TW@^S  + TW@^SE;
      TW := (TW & NN = 2) | (NN = 3);
    until ! (|<< TW);
```

Apply rules to live by

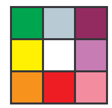
"Or" bits in world to see if any alive

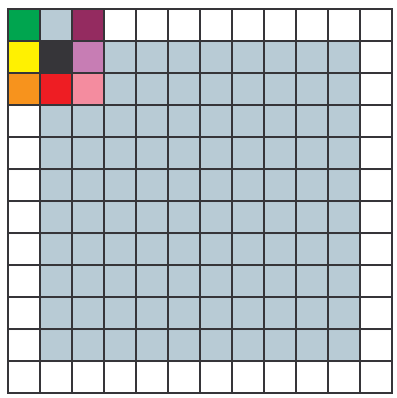
Edges wrap around ⇓



Cartoon of counting neighbors: Array of NW neighbors+ array of north neighbors+array of NE neighbors+...

Stencil Optimizations

- When walking over an array referencing neighbors by stencil,  the references are repeated



Local allocation

Approach:

Recognize stencil usage

Move values to registers

Precompute sums …

Which sums to do?

What can you save?