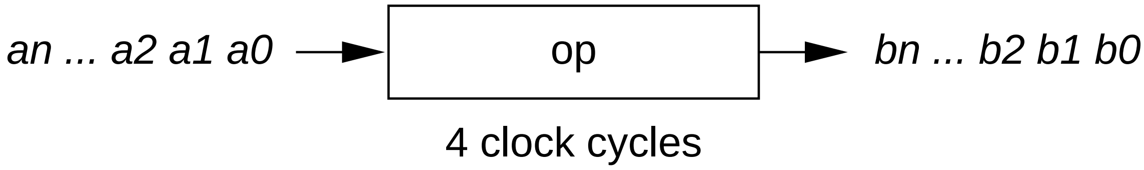
Vector Programming

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

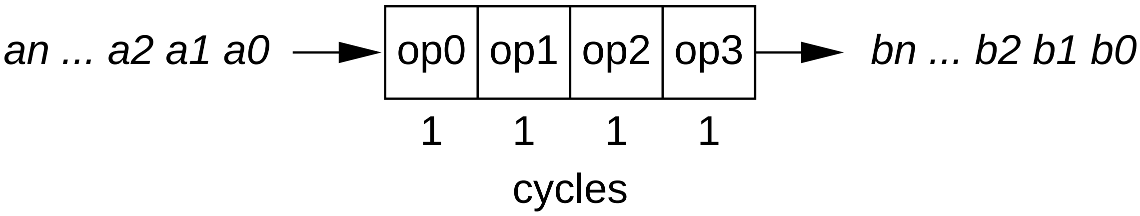
Scalar Processing



Time	op
0	a_0
4	a_1
8	a_2
...	...
$4n$	a_n

$4n$ clock cycles required to process n elements!

Pipelining



Time	op$_0$	op$_1$	op$_2$	op$_3$
0	a_0			
1	a_1	a_0		
2	a_2	a_1	a_0	
3	a_3	a_2	a_1	a_0
4	a_4	a_3	a_2	a_1
...	...	...	...	...
n	a_n	a_{n-1}	a_{n-2}	a_{n-3}

$4n/(4+n)$ *clock cycles required to process* n *elements!*

Basic Principle

- Stream of objects.

  Number of objects = stream length n.

- Operation can be subdivided into sequence of steps.

  Number of steps = pipeline length p.

- Advantage

  Speedup $= pn/(p+n)$

- Stream length $\gg$ pipeline length

  Speedup $\approx p$

Speedup is limited by pipeline length!

Arithmetic Pipeline

For floating point operations

- Floating point = (mantissa, exponent)

  $X = (m_x, e_x) = m_x * 2^{e_x}$
  $Y = (m_y, e_y) = m_y * 2^{e_y}$

- Floating point sum

  $X + Y = (m_x * 2^{e_x - e_y} + m_y) * 2^{e_y}$

Floating point operations dominant in scientific applications!

Arithmetic Pipeline Stages

3 stages for addition

1. Exponent comparison and mantissa adjustment (shift)

   $\boxed{m_x * 2^{e_x - e_y}}$

2. Mantissa addition (add)

   $\boxed{\square + m_y}$

3. Mantissa normalization and exponent readjustment

   $\boxed{\square * 2^{e_y}}$

Problem: How to continuously feed floating point unit (FPU) with stream of operations?

Vector Operations

Operations on vectors of data (floating point numbers)

- Vector-vector

  $V_1 \leftarrow V_2 + V_3$ (component-wise sum)
  $V_1 \leftarrow -V_2$

- Vector-scalar

  $V_1 \leftarrow c * V_2$

- Vector-memory

  $V \leftarrow A$ (vector load)
  $A \leftarrow V$ (vector store)

- Vector reduction

  $c \leftarrow \mathrm{min}(V)$
  $c \leftarrow \mathrm{sum}(V)$
  $c \leftarrow V_1 * V_2$ (dot product)

Vector Operations

- Gather/scatter

  $V_1, V_2 \leftarrow$ GATHER(A)
  load all non-zero elements of A into V_1 and their indices into V_2

  $A \leftarrow$ SCATTER(V_1, V_2)
  store elements of V_1 into A at indices denoted by V_2 and fill rest with zeros

- Mask

  $V_1 \leftarrow$ MASK(V_2, V_3)
  store elements of V_2 into V_1 for which corresponding position in V_3 is non-zero

Scalar Loop

Fortran loop:

```
DO I=1,N
  A(I) = A(I)+B(I)
ENDDO
```

Scalar assembly code:

```
   R0 <- N
   R1 <- I
   JMP J
L: R2 <- A(R1)
   R3 <- B(R1)
   R2 <- R2+R3
   A(R1) <- R2
   R1 <- R1+1
J: JLE R1, R0, L
```

$\approx 6n$ *clock cycles to execute loop.*

Vector Loop

Vectorized assembly code:

```
V1 <- A
V2 <- B
V3 <- V1+V2
A  <- V2
```

$4n$ *clock cycles, because no loop iteration overhead (ignoring speedup by pipelining)*

Chaining

- Overlapping of vector instructions

  (see Hwang, Figure 8.18)

- Hence: $c + n$ ticks (for small c)

  Speedup ≈ 6
  $(c = 16, n = 128, s = (6 * 128)/(16 + 128) = 5.33)$

- The longer the vector chain, the better the speedup!

  $\vec{A} \leftarrow B * \vec{C} + \vec{D}$
  chaining degree 5

- Vectorization speedups between 5 and 25

Vector Programming

How to generate vectorized code?

1. Assembly programming.
2. Vectorized Libraries.
3. High-level vector statements.
4. Vectorizing compiler.

Vectorized Libraries

- Predefined vector operations (partially implemented in assembly language)
  - VECLIB, LINPACK, EISPACK, MINPACK
- `C = SSUM(100, A(1,2), 1, B(3,1), N)`

  `100` . . . vector length

  `A(1,2)` . . . vector address A

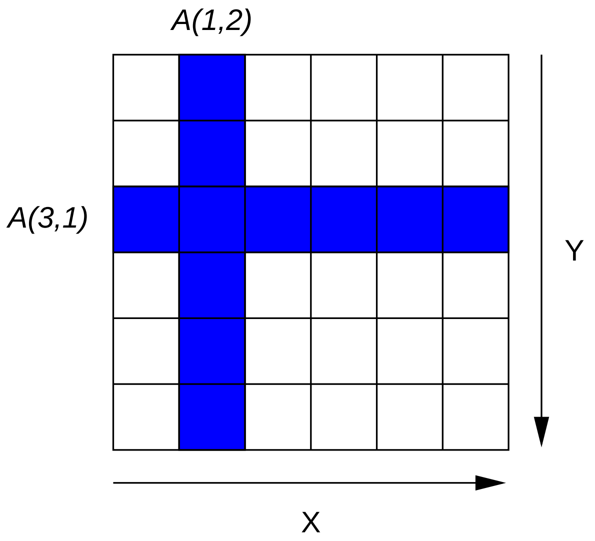
  `1` . . . vector stride A

  `B(3,1)` . . . vector address B

  `N` . . . vector stride B

Addition of matrix column to matrix row.

Fortran Matrices



Stride = distance between vector elements.

- Fortran: "column-major"
- Column stride: 1 word
- Row stride: N words
- Diagonal stride: N+1 words

Vector operations are parameterized by stride; they may be performed on matrix columns, rows, diagonals, ...

High-Level Vector Statements

e.g. Fortran 90

```
INTEGER A(100), B(100), C(100), S
A(1:100) = S*B(1:100)+C(1:100)
```

- Vector-vector operations.
- Vector-scalar operations.
- Vector reduction.
- ...

Easy transformation into vector code.

Vectorizing Compiler

1. Fortran 77 DO Loop
   - ```
     DO I=1, N
       D(I) = A(I)*B+C(I)
     ENDDO
     ```
2. Vectorization
   - ```
     D(1:N) = A(1:N)*B+C(1:N)
     ```
3. Strip mining
   - ```
     DO I=1, N/128
       D(I:I+127) = A(I:I+127)*B + C(I:I+127)
     ENDDO
     IF ((N.MOD.128).NEQ.0)
       A((N/128)*128+1:N) = ...
     ENDIF
     ```
4. Code generation
   - ```
     V0 <- V0*B
     ...
     ```

Related techniques for parallelizing compiler!

Vectorization

In which cases can loop be vectorized?

```
DO I = 1, N-1
  A(I) = A(I+1)*B(I)
ENDDO
```

$$\Downarrow$$

```
A(1:128) = A(2:129)*B(1:128)
A(129:256) = A(130:257)*B(129:256)
....
```

Vectorization preserves semantics.

Loop Vectorization

Is semantics always preserved?

```
DO I = 2, N
  A(I) = A(I-1)*B(I)
ENDDO
```

$$\Downarrow$$

```
A(2:129) = A(1:128)*B(2:129)
A(130:257) = A(129:256)*B(130:257)
....
```

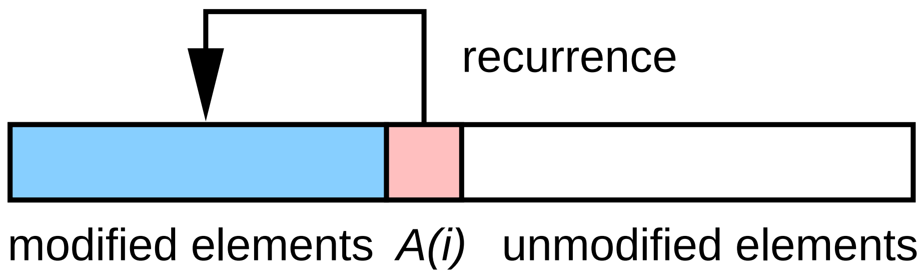
Vectorization has changed semantics!

Vectorization Inhibitors

Vectorization must be conservative; when in doubt, loop must *not* be vectorized.
Vectorization is inhibited by

- Function calls
- Input/output operations
- GOTOs into or out of loop
- Recurrences

  References to vector elements modified in previous iterations.



Goal is to maximize vectorization!

Vectorization Principles

- Loops over arrays

```
DO I = ...
  ... A(I) ...
ENDDO
```

- Induction variables

  Incremented/decremented by constant during each iteration

```
J = 1
DO I = 1, N
  J = J+3
  A(I) = A(J)*B(I)
ENDDO
```

  `I` and `J` are induction variables

Vectorization Principles

- Subscripts linear in induction variable

  Linear combination of induction variable and constants

```
DO I = ...
  ... A(I+C) ... A(D*I+C) ...
ENDDO
```

- Only innermost loop is vectorized

```
DO I = ...
  DO J = I+1, ...
    B(J-I) = A(I)*B(J)
  ENDDO
ENDDO
```

  I is constant in inner loop!

Compiler transforms loops to allow vectorization and/or to improve performance.

Partial Vectorization

Split loops to allow vectorization take place.

Non-vectorizable loop

```
DO I = 1, N
  A(I) = A(I-1)*X
  B(I) = C(I)**Y
ENDDO
```

$$\Downarrow$$

Non-vectorizable loop

```
DO I = 1, N
  A(I) = A(I-1)*X
ENDDO
```

Vectorizable loop

```
DO I = 1, N
  B(I) = C(I)**Y
ENDDO
```

Loop Interchange

Interchange nested loops

```
DO I = 1, N
  DO J = 1, N
    ...
  ENDDO
ENDDO
```

$$\Downarrow$$

```
DO J = 1, N
  DO I = 1, N
    ...
  ENDDO
ENDDO
```

- Increase iteration count
- Achieve most efficient vector stride (+1)
- Avoid non-vectorizable recurrences

Loop Distribution

Allow code in outer loop to be vectorized.

```
DO J = 1, N
  A(J,J) = X(J)*3
  DO I = 1, N
    B(I,J) = A(I,J)*C(I,J)
  ENDDO
ENDDO
```

$$\Downarrow$$

```
DO J = 1, N
  A(J,J) = X(J)*3
ENDDO

DO J = 1, N
  DO I = 1, N
    B(I,J) = A(I,J)*C(I,J)
  ENDDO
ENDDO
```

Both loops can be vectorized!

Vectorization of Conditionals

Merge different vectors.

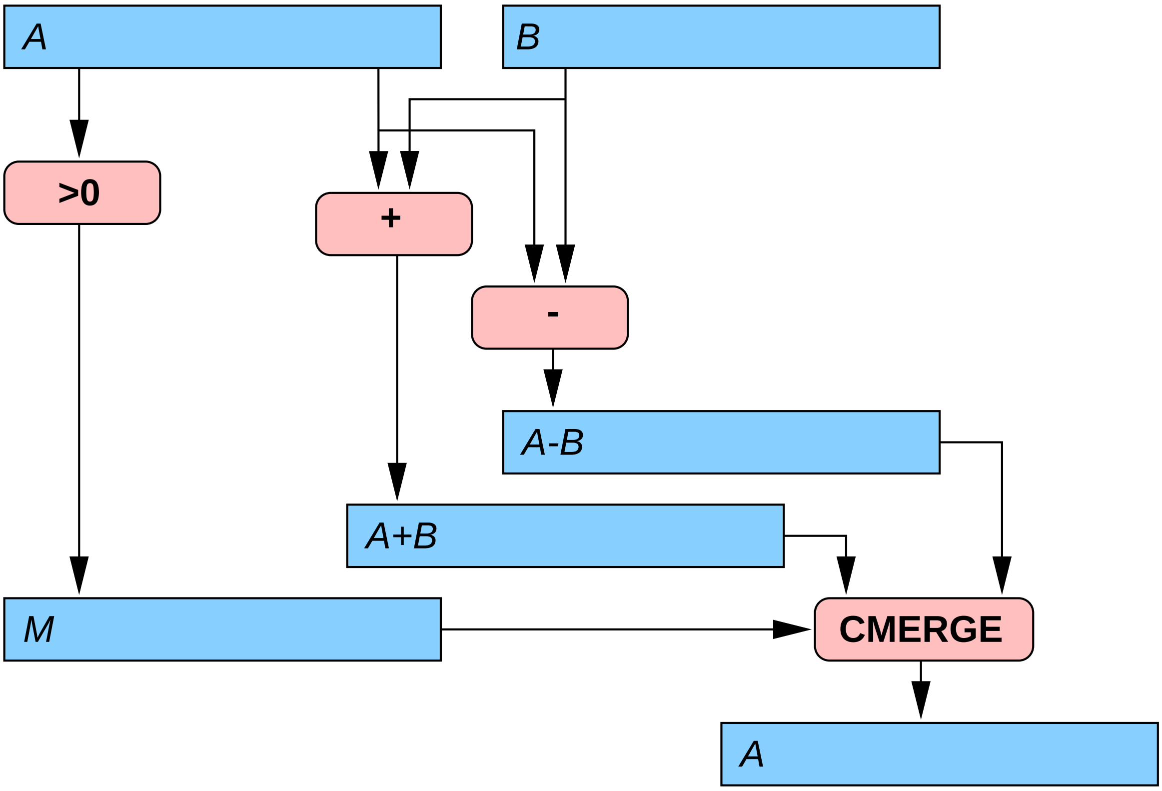
```
DO I = 1, N
  IF (A(I).GT.0)
    A(I) = A(I)+B(I)
  ELSE
    A(I) = A(I)-B(I)
  ENDIF
ENDDO
```

$$\Downarrow$$

```
M(1:N) = A(1:N).GT.0
A(1:N) = CMERGE(M, A+B, A-B)
```

Both branches are computed for all elements!

Conditional Vector Merge



Vector Reductions

Also some assignments into scalars can be vectorized.

```
S = 0
DO I = 1, N
  S = S+A(I)
ENDDO
```

$$\Downarrow$$

```
S = VECTOR_SUM(A)
```

Only for predefined reduction operations!

Example

Matrix multiplication

Fortran 77 source program

```
DO I = 1, N
  DO J = 1, N
    C(I,J) = 0
    DO K = 1, N
      C(I,J) = C(I,J)+A(I,K)*B(K,J)
    ENDDO
  ENDDO
ENDDO
```

(See additional slide)

Loop Distribution

Loop 4 ⇒ loop 4-1 and loop 4-2

```
4-1 DO I = 1, N
5-1   DO J = 1, N
6-1     C(I,J) = 0
      ENDDO
    ENDDO

4-2 DO I = 1, N
5-2   DO J = 1, N
7-2     DO K = 1, N
8-2       C(I,J)=C(I,J)+A(I,K)*B(K,J)
        ENDDO
      ENDDO
    ENDDO
```

Loop 5-1 becomes vectorizable!

Loop Interchange

Interchange loop 4-1 and loop 5-1

```
5-1 DO J = 1, N
4-1   DO I = 1, N
6-1     C(I,J) = 0
      ENDDO
    ENDDO
```

Vector stride becomes one word!

Interchange loop 5-2 and loop 7-2

```
5-2 DO J = 1, N
7-2   DO K = 1, N
4-2     DO I = 1, N
8-2       C(I,J)=C(I,J)+A(I,K)*B(K,J)
        ENDDO
      ENDDO
    ENDDO
```

Type of vector operation is changed!

Types of Vector Operations

```
DO _ = 1, N
  C(I,J) = C(I,J)+A(I,K)*B(K,J)
ENDDO
```

- Index K

  I and J are fixed
  $\Rightarrow$ Scalar = Scalar + Vector$*$Vector
  $\Rightarrow$ 1 vector reduction!

- Index J

  I and K are fixed
  $\Rightarrow$ Vector = Vector + Scalar$*$Vector
  $\Rightarrow$ 1 vector-vector, 1 scalar-vector
  $\Rightarrow$ vector strides n

- Index I

  J and K are fixed
  $\Rightarrow$ Vector = Vector + Vector$*$Scalar
  $\Rightarrow$ 1 vector-vector, 1 scalar-vector
  $\Rightarrow$ vector strides 1

Transformation Result

```
5-1 DO J = 1, N
4-1   DO I = 1, N
6-1     C(I,J) = 0
      ENDDO
    ENDDO
5-2 DO J = 1, N
7-2   DO K = 1, N
4-2     DO I = 1, N
8-2       C(I,J) = C(I,J)+A(I,K)*B(K,J)
        ENDDO
      ENDDO
    ENDDO
```

- Vector loops 4-1 and 4-2
- Scalar loops 5-1, 5-2 and 7-2

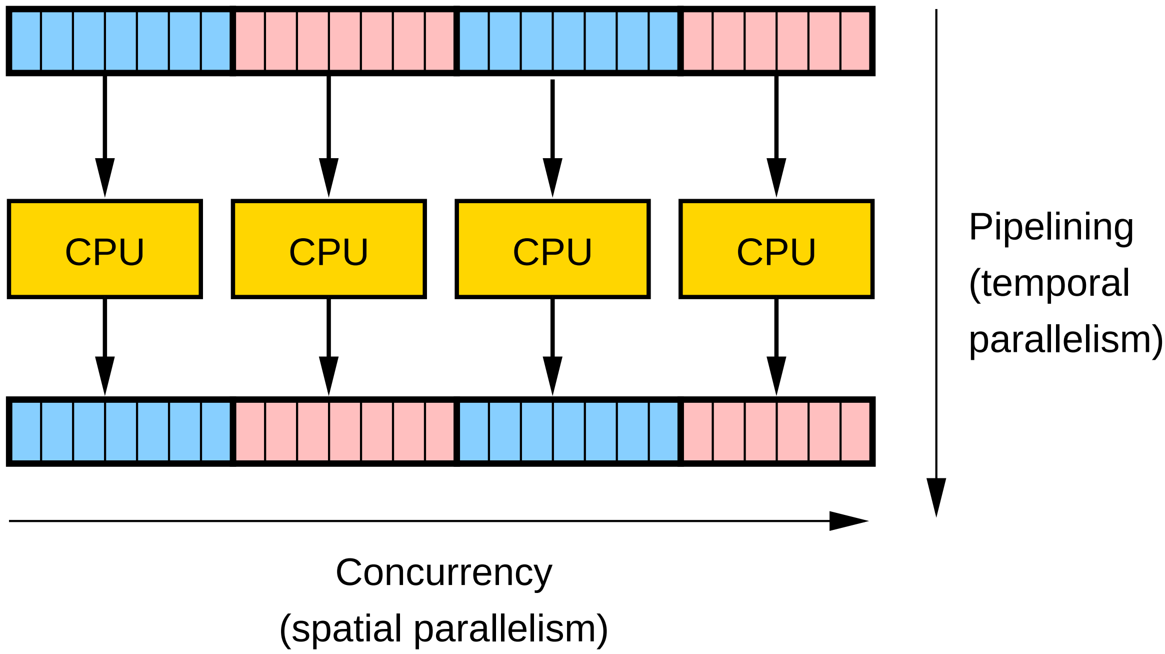
Vectorization Result

```
5-1 DO J = 1, N
      C(1:N,J) = 0
    ENDDO
5-2 DO J = 1, N
7-2   DO K = 1, N
        C(1:N,J)=C(1:N,J)+A(1:N,K)*B(K,J)
      ENDDO
    ENDDO
```

- Vector sources ("hoists") A, C
- Vector target ("sunk") C

Vector-Multiprocessors

Several vector processors operate on shared memory.



- Pipelining within processor.
- Concurrency among processors.

Example: Convex C3440