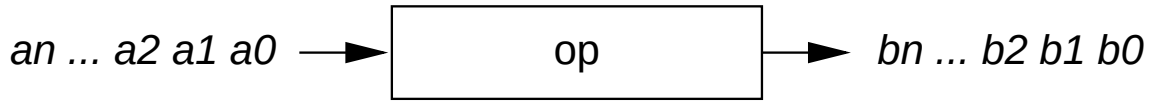
Vectorization I

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

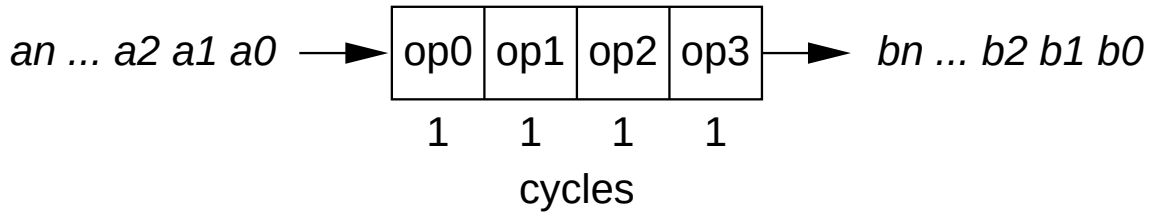


4 clock cycles

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!

Instruction Pipeline

- All modern micro-processors,
- Stream of instructions.

(see Hwang, Figure 6.9)

$$
\begin{array}{ll}
X = Y + Z \Rightarrow & R_1 \leftarrow Y \\
A = B * C & R_2 \leftarrow Z \\
& R_4 \leftarrow B \\
& R_5 \leftarrow C \\
& R_3 \leftarrow R_1 + R_2 \\
& R_6 \leftarrow R_4 * R_5 \\
& X \leftarrow R_3 \\
& A \leftarrow R_6
\end{array}
$$

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

  $c \leftarrow \min(V)$
  $c \leftarrow \text{sum}(V)$
  $c \leftarrow V_1 * V_2$ (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 Architectures

Vector Processor consists of vector registers and vector units.

- Convex C3440 (4 processors)

8 registers	V0–V7	3 units:	load/store
a	128 words		multiply/divide
a	64 bit		add/logical

- Cray C90 (16 processors)

8 registers	V0–V7	11 units:	2 load
a	64 words		1 store
a	64 bit		8 arithmetic

Vector Programming

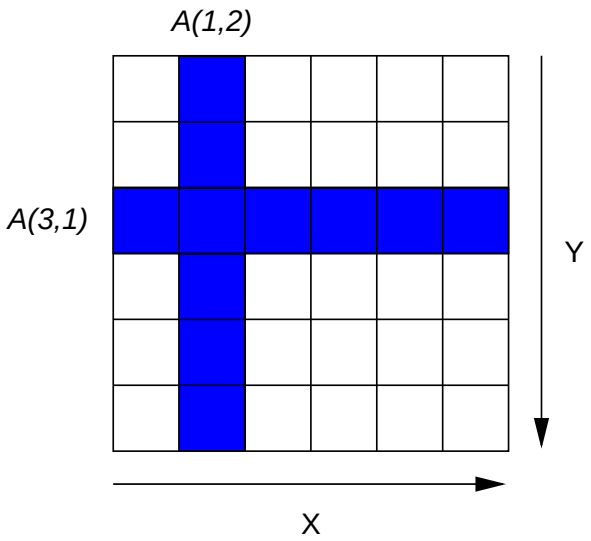
How to generate vector instructions?

- Use vectorized library
- 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)
```

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
  ...
  ```

Vectorization

- Vectorization
  - Detect "vector" pattern in sequential loop.
- Strip mining
  - Adjust vector instructions to actual length of vector registers.

Vectorization is conservative i.e. it must not change the semantics of the program!