Basic rules to achieve vector performance

- Raising Vectorisation Ratio

  = number of vector instructions / number of execution instructions

- Improving Vector Instruction Efficiency

  = effective vector power / maximum (peak) vector power

Empowered by Innovation NEC

Improving vector instruction efficiency

…by lengthening the loops

Vectorisation start-up time is independent on loop length
Short loops significantly reduce the vectorisation efficiency

→ Maximize the length of the innermost loop

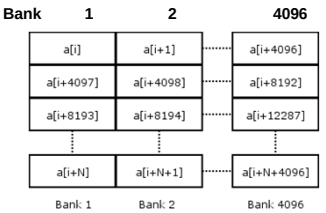
Example :

```
N = 10000
M = 10
! CDIR LOOPCHG
DO J=1,N
  DO I=1,M
    A(I,J) = X*B(I,J)+C(I,J)
  END DO
END DO
```

```
DO I=1,M
  DO J=1,N
   A(I,J) = X*B(I,J)+C(I,J)
  END DO
END DO
```


…by improving array reference patterns

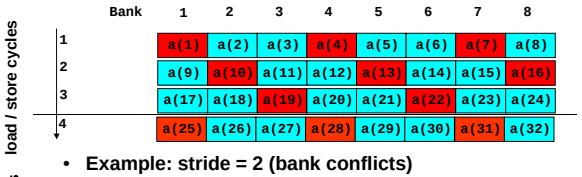
- Vector loading and storing speed is highest when the interval between consecutive elements is an odd constant number:
  stride = 1, 3, 5, 7, …
- Consider SX Shared Memory with 32 x 128 = 4096 memory banks and vector a(:) :



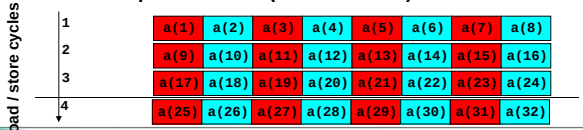
cf. SUPER-UX Performance Tuning Guide

…by improving array reference patterns

Super simplified example
- 8 banks
- 3 clock cycles to LOAD or STORE an a(i)
  - Example: stride = 3 (no bank conflicts)



  - Example: stride = 2 (bank conflicts)



Effect of bank conflicts: SX-6

```
DO i = 1,n,stride
   a(i) = s + b(i)
END DO
```

stride	time [s]	performance [MFlops]	factor
1	5.927E-05	1089.18	1.00
2	7.512E-05	1331.20	1.49
3	5.823E-05	1980.76	1.00
4	1.413E-04	707.69	2.81
5	5.832E-05	1091.00	1.00
6	7.512E-05	1331.20	1.49
8	2.915E-04	343.08	5.80
10	7.512E-05	1331.20	1.49
12	1.413E-04	707.69	2.81
16	6.820E-04	146.62	13.57
24	2.914E-04	343.15	5.80
32	1.485E-03	67.33	29.54
48	6.840E-04	146.20	13.61
64	1.485E-03	67.33	29.54
128	1.486E-03	67.31	29.55
256	1.501E-03	66.60	29.87
		36.44	54.59
		27.84	71.46
2048	6.761E-03	14.79	134.50
4096	1.344E-02	7.44	267.28
4097	5.042E-05	1983.18	1.00
8192	1.344E-02	7.44	267.28

Bank conflicts when stride is even

…by improving array reference patterns

Wherever possible, vectorise along the first dimension

Example

```
REAL, DIMENSION (100,100) :: A, B, C
 …
DO I=1,N
  DO J=1,N
    A(I,J) = B(I,J) + X*C(I,J)
  END DO
END DO
```

↓

```
REAL, DIMENSION (100,100) :: A, B, C
 …
DO J=1,N
  DO I=1,N
    A(I,J) = B(I,J) + X*C(I,J)
  END DO
END DO
```


…by increasing concurrency

Vector +, -, *, /, shift and logical operations can be executed in parallel

→ Put as many of these operations together in one loop as possible

Example

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

DO I=1,N
     X(I) = Y(I)*Z(I)
END DO
```

→

```
DO I=1,N
     A(I) = B(I)+C(I)
     X(I) = Y(I)*Z(I)
END DO
```


…by converting divisions into multiplications

Reason: vector division is slower than other vector arithmetic operations

Example

```
DO i=1,1000
  a(i) = b(i)/value
END DO
```

→

```
temp = 1/value
DO i=1,1000
  a(i) = b(i)*temp
END DO
```


…by using scalar temporary variables instead of arrays

Example

```
DO I=1,N
   WX(I) = A(I)+B(I)
   WY(I) = C(I)-D(I)
   E(I)  = S*WX(I)+T*WY(I)
   F(I)  = S*WY(I)-T*WX(I)
END DO
```

→

```
DO I=1,N
   X    = A(I)+B(I)
   Y    = C(I)-D(I)
   E(I) = S*X+T*Y
   F(I) = S*Y+T*X
END DO
```

Inefficient because additional store operations are necessary

`-C hopt` creates work-arrays in simple cases

…by reducing the number of instructions

Original

```
vhmx_vol = 0.0
DO k = 1,ke
   DO j = jstartu,jendu
      DO i = istartv,iendv
         zvb = SQRT ( u(i,j,k,nnow)**2 +v(i,j,k,nnow)**2 )
         vhmx_vol = MAX ( vhmx_vol, zvb)
      ENDDO
   ENDDO
ENDDO
zuvwmax(0) = vhmx_vol
```

Optimised

```
vhmx_vol = 0.0
DO k = 1,ke
   DO j = jstartu,jendu
      DO i = istartv,iendv
         zvb = u(i,j,k,nnow)**2 + v(i,j,k,nnow)**2
         vhmx_vol = MAX ( vhmx_vol, zvb)
      ENDDO
   ENDDO
ENDDO
zuvwmax(0) = SQRT (vhmx_vol)
```


Basic rules to achieve vector performance

- Raising Vectorisation Ratio
  - remove the cause of non-vectorization !
- Improving Vector Instruction Efficiency
  - Long loops
  - Stride 1 memory access
  - Vectorisation along 1st dimension
  - Concurrency
  - …

Empowered by Innovation NEC

Code tuning – “Loop pushing”

Loop pushing

```
subroutine work1(t)
use data
integer i,j,t

do j=1,je
  do i=1,ie
    do k=1,ke
      local1(k)=dble(t*i+j+k)
      a(i,j,k)=b(i,j,k)+local1(k)
    end do
    call work2(i,j)
    do k=1,ke
      a(i,j,k)=b(i,j,k)+local2(k)
    end do
  end do
end do
end subroutine
```

Performance bottlenecks:

1. Many subroutine calls in loop nest.
2. Vectorization of loops working on outer dimension.

Loop pushing

```
subroutine work2(i,j)
use data
integer i,j,k
do k=1,ke
  b(i,j,k)=sin(dble(i))+cos(dble(j))+tan(dble(k))
end do
local2(1)=1
local2(ke)=1
do k=2,ke-1
  local2(k)=dble(i+j)/dble(k)+local1(k)
end do
end subroutine
```

Performance bottlenecks:

1. Pointwise work on (short?) loop on 3rd dimension.

Loop pushing – “How to”

```
subroutine work1(t)
use data
integer i,j,t

do j=1,je
  do i=1,ie
    do k=1,ke
      local1(k)=dble(t*i+j+k)
      a(i,j,k)=b(i,j,k)+local1(k)
    end do
    call work2(i,j)
    do k=1,ke
      a(i,j,k)=b(i,j,k)+local2(k)
    end do
  end do
end do
end subroutine
```

Goal: Separate “calculation loops” and subroutine calls.

Loop pushing – "How to"

```
subroutine work1(t)
use data
integer i,j,t

do j=1,je
  do i=1,ie
    do k=1,ke
      local1(i,j,k)=dble(t*i+j+k)
      a(i,j,k)=b(i,j,k)+local1(i,j,k)
    end do



    call work2(i,j)



    do k=1,ke
      a(i,j,k)=b(i,j,k)+local2(i,j,k)
    end do
  end do
end do
end subroutine
```

1. Promote variables where necessary (also in subroutines called in the loop).

Loop pushing – "How to"

```
subroutine work1(t)
use data
integer i,j,t

do j=1,je
  do i=1,ie
    do k=1,ke
      local1(i,j,k)=dble(t*i+j+k)
      a(i,j,k)=b(i,j,k)+local1(i,j,k)
    end do
  end do
end do
do j=1,je
  do i=1,ie
    call work2(i,j)
  end do
end do
do j=1,je
  do i=1,ie
    do k=1,ke
      a(i,j,k)=b(i,j,k)+local2(i,j,k)
    end do
  end do
end do
end subroutine
```

1. Promote variables where necessary (also in subroutines called in the loop).

2. Cut into separate loops to isolate subroutine calls.

Loop pushing – "How to"

```
subroutine work1(t)
use data
integer i,j,t

do j=1,je
  do i=1,ie
    do k=1,ke
      local1(i,j,k)=dble(t*i+j+k)
      a(i,j,k)=b(i,j,k)+local1(i,j,k)
    end do
  end do
end do

    call work2

do j=1,je
  do i=1,ie
    do k=1,ke
      a(i,j,k)=b(i,j,k)+local2(i,j,k)
    end do
  end do
end do
end subroutine
```

1. Promote variables where necessary (also in subroutines called in the loop).

2. Cut into separate loops to isolate subroutine calls.

3. Push loop(s) into subroutine.

A working version is available after each step!

Loop pushing – "How to"

```
subroutine work1(t)
use data
integer i,j,t

do k=1,ke
  do j=1,je
    do i=1,ie
      local1(i,j,k)=dble(t*i+j+k)
      a(i,j,k)=b(i,j,k)+local1(i,j,k)
    end do
  end do
end do

    call work2

do k=1,ke
  do j=1,je
    do i=1,ie
      a(i,j,k)=b(i,j,k)+local2(i,j,k)
    end do
  end do
end do
end subroutine
```

1. Promote variables where necessary (also in subroutines called in the loop).

2. Cut into separate loops to isolate subroutine calls.

3. Push loop(s) into subroutine.

4. Change loop orders.

A working version is available after each step!

Loop pushing – “How to”

```
subroutine work2
use data
integer i,j,k
do j=1,je
  do i=1,ie
    do k=1,ke
      b(i,j,k)=sin(dble(i))+cos(dble(j))+tan(dble(k))
    end do
    local2(i,j,1)=1
    local2(i,j,ke)=1
    do k=2,ke-1
      local2(i,j,k)=dble(i+j)/dble(k)+local1(i,j,k)
    end do
  end do
end do
end subroutine
```

1. Loop(s) are pushed into the subroutine.

Still a working code version!

Loop pushing – “How to”

```
subroutine work2
use data
integer i,j,k
do k=1,ke
  do j=1,je
    do i=1,ie
      b(i,j,k)=sin(dble(i))+cos(dble(j))+tan(dble(k))
      local2(i,j,k)=dble(i+j)/dble(k)+local1(i,j,k)
    end do
  end do
end do
do j=1,je
  do i=1,ie
    local2(i,j,1)=1
    local2(i,j,ke)=1
  end do
end do
end subroutine
```

1. Change loop order to allow for vectorization of “right” loops.

Loop pushing - effect

- Module variables **`a, b, local1, local2`**
- Dimensions **`i = 250, j = 200, k = 25`**
- **`work1`** called 300 times from driver program

- Original code version: 12.9 seconds real time

- Optimized version with loop pushing and reordering:
  0.28 seconds

- Run time reduction by a factor of 46.

