

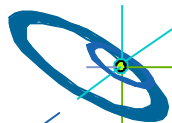


Tutorial

Using Vector Registers (VREGs) on NEC SX-8

Katharina Benkert
benkert@hirs.de

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hirs.de



Contents



1. A sample problem
2. Use of vector registers

The fortran code snippets can be downloaded from my webpage:

<http://www.hlrs.de/people/benkert>



A sample problem (sub_simple)

Example: Compute two matrix-vector products with same matrix A, but different right hand sides

```
s = A * t
x = A * y
```

```
subroutine sub_simple(
!   ... Declarations
```

```
s(:) = 0.0; x(:) = 0.0
do j = 1, n
  do i = 1, n
    s(i) = s(i) + A(i,j) * t(j)
    x(i) = x(i) + A(i,j) * y(j)
  end do
end do
```

```
end subroutine sub_simple
```

Compiler messages:

```
f90: opt(1800): vreg_example.f90:
      Idiom detected (matrix multiply).
f90: opt(1800): vreg_example.f90:
      Idiom detected (matrix multiply).
```

A sample problem: What is the compiler doing? (sub_guess)

```
subroutine sub_guess(n, l, s, t, x, y, A)
!   ... Declarations ...
do i = 1, n
    sx(i,1) = 0.0_dp; sx(i,2) = 0.0_dp
    ty(i,1) = t(i);   ty(i,2) = y(i)
end do
do i = 1, n
    do j = 1, l
        do k = 1, n
            sx(i,j) = sx(i,j) + A(i,k) * ty(k,j)
        end do
    end do
end do
do i = 1, n
    s(i) = sx(i,1);   x(i) = sx(i,2)
end do
end subroutine sub_guess
```

Just a guess!



A sample problem: What is the compiler doing?

If the compiler's solution is not a 100% optimal, we might end up

- loading the matrix A separately for each matrix-vector multiplication from main memory

and / or

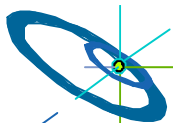
- using more than one load and store for each of the result vectors s and x

This is our chance to apply vector registers. They provide low-level access to the underlying hardware and we are therefore able to exert some control over memory access.



Contents

1. A sample problem
2. Using vector registers



Stripmining – A necessary prerequisite

Since the length of vector registers, the hardware vector length VL , is currently limited to $256 * 8$ Bytes, we have to apply stripmining, i.e. split up arrays into blocks of 256 elements.

This loop for example

```
for i = 1, n
  s(i) = 0.0
end do
```

Directive to tell the compiler that this loop is only 256 elements long and that it doesn't have to stripmine the loop itself

becomes

```
do i0 = 1, n, VL
  !cdir shortloop
  do i = i0, min(n, i0+VL-1)
    s(i) = 0.0
  end do
end do
```

Vector registers - Declaration

```
! declare hardware vector length as parameter, it might
! be different for the next generation of vector
! computers
integer, parameter :: VL = 256

! declare 2 double precision arrays of size VL
real*8, dimension(VL) :: vr_s, vr_x

! tell the compiler that they should be some vector
! registers
!cdir vreg(vr_s, vr_x)
```



Vector registers - The typical usage

```
do i0 = 1, n, VL ! outermost loop
```

```
!cdir shortloop
```

```
do i = i0, min(n, i0+VL-1)
```

```
vr(i-i0+1) = 0.0d0
```

```
end do
```

Initialization

```
... more loops ...
```

```
!cdir shortloop
```

```
do i = i0, min(n, i0+VL-1)
```

```
vr(i-i0+1) = ....
```

```
end do
```

**innermost loop where
vector registers are used;
watch out that you access
only elements 1 to 256.**

```
... end of more loops ...
```

```
!cdir shortloop
```

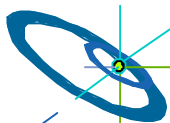
```
do i = i0, min(n, i0+VL-1)
```

```
.... = vr(i-i0+1)
```

```
end do
```

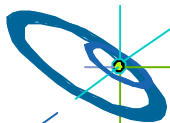
store results

```
end do
```



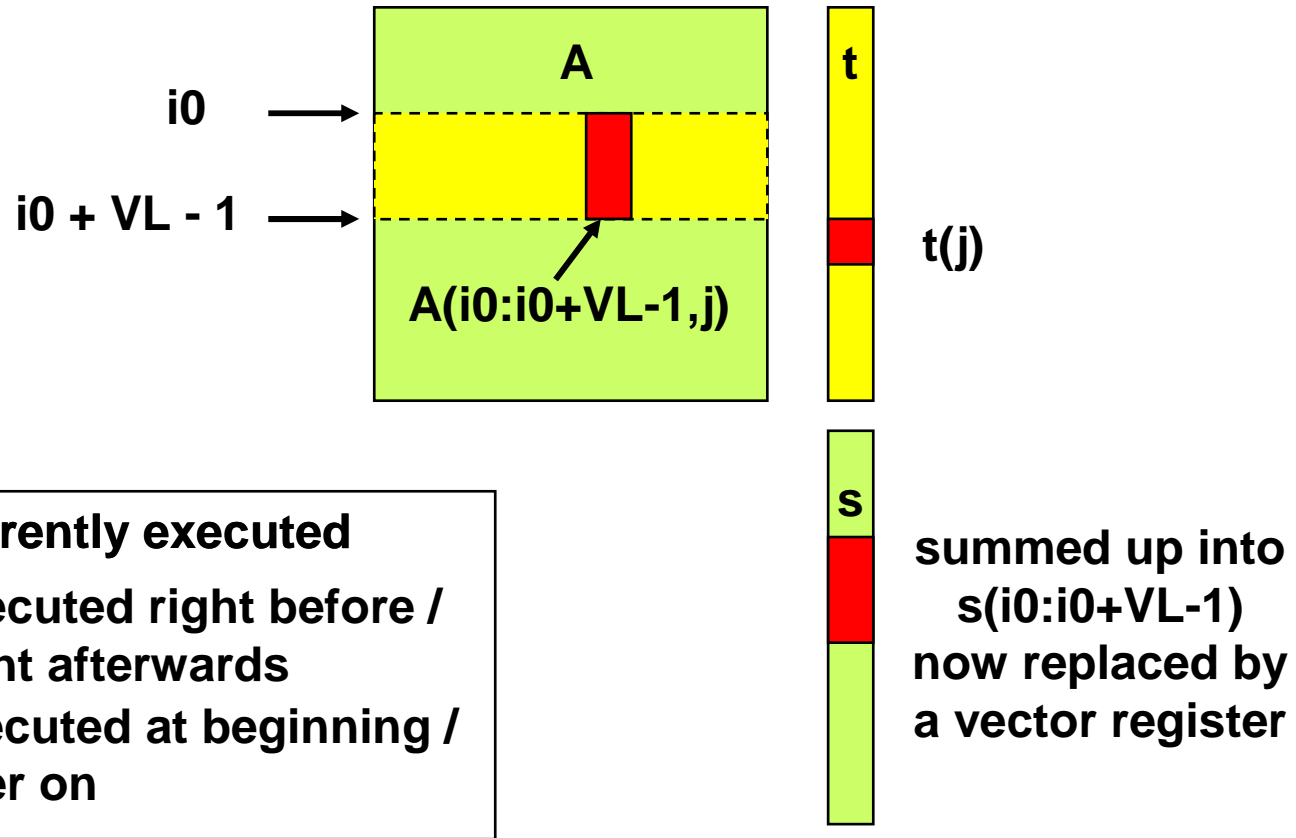
Vector registers for our example (sub_vreg)

```
do i0 = 1, n, VL
!cdir shortloop
  do i = i0, min(n, i0+VL-1)
    vr_s(i-i0+1) = 0.0d0;    vr_x(i-i0+1) = 0.0d0
  end do
  do j = 1, n
!cdir shortloop
    do i = i0, min(n, i0+VL-1)
      vr_s(i-i0+1) = vr_s(i-i0+1) + A(i,j) * t(j)
      vr_x(i-i0+1) = vr_x(i-i0+1) + A(i,j) * y(j)
    end do
  end do
!cdir shortloop
  do i = i0, min(n, i0+VL-1)
    s(i) = vr_s(i-i0+1);    x(i) = vr_x(i-i0+1)
  end do
end do
```



Vector registers for our example

Graphical representation



Vector registers – How to transform your code

Suppose the loop variable for the vector registers is i

1. Copy your loop in i as an outer loop, change i to i_0 and add an increment of VL
2. Change inner loop $i = 1, n$ into $i = i_0, \min(n, i_0+VL-1)$
3. Declare the vector register(s)
4. Add an initialization loop
5. Use vregs for the innermost loop
6. Copy results back

Check, if

- Your index shift when accessing elements in the vector registers is correct
- You specified `shortloop` compiler directives where necessary



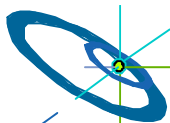
Vector registers – Results for our example

Extract of ftrace output for the three subroutines presented:
n=50, nloops=1.000.000

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	MFLOPS	V.OP RATIO	AVER. V.LEN
example_vreg					
	1	38.167(74.3)	315.0	94.07	106.8
sub_guess	1000000	4.773(9.3)	8380.0	98.77	100.0
sub_simple	1000000	4.449(8.7)	8990.6	98.69	100.0
sub_vreg	1000000	3.996(7.8)	10008.9	98.46	111.9

total	3000001	51.386(100.0)	2569.3	96.83	105.3

So our guess, what the compiler might be doing, is wrong, but:
The version using vector registers is faster than the compiler
generated code!



However this version ...

```
subroutine sub_2mult(n, s, t, x, y, A)
!   ... Declarations ...

s(:) = 0.0; x(:) = 0.0
do j = 1, n
  do i = 1, n
    s(i) = s(i) + A(i,j) * t(j)
  end do
end do

do j = 1, n
  do i = 1, n
    x(i) = x(i) + A(i,j) * y(j)
  end do
end do
end subroutine sub_2mult
```



... is faster

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec](%)	MFLOPS	V.OP RATIO	AVER. V.LEN
example_vreg	1	38.503(76.2)	312.3	94.07	106.8
sub_simple	1000000	4.459(8.8)	8971.0	98.69	100.0
sub_vreg	1000000	3.953(7.8)	10118.4	98.46	111.9
sub_2mult	1000000	3.582(7.1)	11165.7	98.48	100.0

total	3000001	50.497(100.0)	2614.5	96.77	105.3

– The End –

Please send questions and remarks to benkert@hls.de.

