Vektorisierungsbeispiele

Uwe Küster

Vektorisierung

- automatisch durch Compiler erzeugt
- Pipelining und parallele functional units
- auch: software pipelining
- Unterstützung durch
  - geeignete Algorithmen
  - geeignete Formulierung / Codierung

Unterschied der Fragestellung

kann ein Programm, ein Unterprogramm, eine Schleife parallelisiert werden?

warum wird eine Schleife nicht vektorisiert?

Examples for vectorizable and non vectorizable loops

The simplest loop

```
      subroutine simple_add(a,b,c,n)
      real a(n),b(n),c(n)
***    vectorizable

      do 10 i=1,n
        a(i)=b(i)+c(i)
10    continue

      return
      end
```


Linear recursion

```
      subroutine linear_recursion(a,b,c,n)
      real a(n),b(n),c(n)
***    non vectorizable recursion
***    better performance with hardware shortcut

      do 10 i=2,n
      a(i)=b(i)*a(i-1)+c(i)
10    continue

      return
      end
```

no recursion

```
      subroutine no_recursion(a,b,c,n)
      real a(n),b(n),c(n)
*** vectorizable
*** old value is used
*** overwritten by new value

      do 10 i=1,n
      a(i)=b(i)*a(i+1)+c(i)
10    continue

      return
      end
```


conditional recursion 1

```
subroutine constant_propagation(a,b,n)
integer n
real a(*),b(*)
call conditional_recursive(a,b,n-1,+1) ! Not recursive
call conditional_recursive(a,b,n-1,-1)  ! recursive
return
end
subroutine conditional_recursive(a,b,n,m)
integer n,m
real a(*),b(*)
do i = 1,n
   a(i) = a(i+m) + b(i)
enddo
return
end
```

conditional recursion 2

```
      subroutine conditional_vectorizable(a,b,c,n)
      real a(n,n),b(n,n),c(n,n)
***    vectorizable if n1/=n2

      n1=n/2
      n2=n
      do 10 i=2,n
      a(i,n1)=a(i-1,n2)+b(i,1)
10    continue

      return
      end
```


unclear dependencies

```
      subroutine constant_index_for_same_vector(a,b,k,n)
      real a(n),b(n),c(n)
***    vectorizable for decomposition
***    in part <k and part >k

      do 10 i=1,n
      a(i)=a(k)+b(i)
10    continue

      return
      end
```

Temporary scalars

```
      subroutine temporary_scalars(a,b,c,n)
      real a(n),b(n),c(n)
***    usage of temporary scalars
***    vectorizable
***    d1,d2 are private in parallel context

      do 10 i=1,n
      d1=b(i)**2+c(i)**2
      d2=b(i)**2-c(i)**2
      a(i)=d2/d1
10    continue

      return
      end
```


Reduction

```
      subroutine scalar_product(alpha,a,b,n)
      real a(n),b(n),c(n)
***    vectorizable reduction
***    special algorithm
***    n log n   complexity

      alpha=0.
      do 10 i=1,n
        alpha=alpha+a(i)*b(i)
10    continue

      return
      end
```

another recursion

```
      subroutine reduction_assignment(a,b,c,n)
      real a(n),b(n),c(n)
***   Kombination aus Reduktion und Zuweisung
***   vectorizable with hardware shortcut
***   on NEC

        sum=0.
      do 10 i=1,n
        sum=sum+a(i)
        b(i)=sum+c(i)
10    continue

      return
      end
```


scalar for next iteration

```
      subroutine scalar_for_next_iteration(a,b,c,d,n)
      real a(n),b(n),c(n),d(n)
***   vectorizable

      s=0.
      do 10 i=1,n
      a(i)=s+b(i)
      s=c(i)+d(i)
10    continue

      return
      end
```

Inlining

```
      subroutine loop_with_procedure_call(a,c,d,n)
      real a(n),c(n),d(n)

      do 10 i=1,n
         a(i)=c(i)+d(i)

*CDIR IEXPAND or f90 -pi noauto  exp=funk ....
         call funk(a(i),n)

10    continue
      return
      end

      subroutine funk(a,n)
      a=a+real(n)
      return
      end
```


overloaded operator +: definition

```
module test_module

type var_type
     real(kind=real_kind)  :: var
end type var_type

interface operator (+)
    module procedure var_add_var
end interface
contains
function var_add_var(aa,bb) result(result_value)
type(var_type),intent(in)  :: aa,bb
type(var_type)                :: result_value
    result_value%var=aa%var+bb%var
end function var_add_var
end module test_module
```

overloaded operator +: usage

```
use test_module

type(var_type),dimension(mmx) :: aa,bb,cc

 do mm=1,mmax
   aa(mm)=bb(mm)+cc(mm)
  enddo
```

inlining is essential


```
      subroutine increment_by_unchanged_value(a,b,c,d,n)
      real a(n),b(n),c(n),d(n)
***    vectorizable

      do 10 i=1,n
      a(i)=b(i)+c(i)
      d(i)=d(i)+a(i+1)
10    continue

      return
      end
```

complicated indices

```
      subroutine calculated_indices(a,b,c,n)
      real a(n),b(n),c(n)
***   complicated index expressions
***    vectorizable under special conditions

      ns=int(sqrt(real(n)))
      do 10 i=1,ns
      j=i*i
      k=i/4
      a(j)=b(k)+c(i)
10    continue

      return
      end
```


unnecessary if clause

```
      subroutine ConditionalChangeOfSingleValue(a,b,c,d,n)
      real a(n),b(n),c(n),d(n)
***    bad Code

      do 10 i=1,n
      if(i.gt.1) then
        c(i) =b(i)-b(i-1)
      else
        c(1) =b(2)-b(1)        ! better:  c(i)=b(2)-b(1)
      endif
10     continue

      return
      end
```

indirect addressing on right side

```
      subroutine sparse_vector_triad(a,b,c,index,n)
      real a(n),b(n),c(n)
      integer index(n)
***    vectorizable
***   slow on cache machines
***   for large extent of  index

      do 10 i=1,n
      a(i)=a(i)+b(i)*c(index(i))
10    continue

      return
      end
```


random accumulate

```
      subroutine histogramm_loop(a,b,index,n)
      real a(n),b(n)
      integer index(n)
***    not vectorizable in former times
***    now vectorizable by masking techniques
***    but slow
***    addition of stiffness matrix in Finite Elements

      do 10 i=1,n
      a(index(i))=a(index(i))+b(i)
10    continue

      return
      end
```

random accumulate with unique index vector

```
      subroutine histogramm_loop_with_directive(a,b,index,n)
      real a(n),b(n)
      integer index(n)
***    if index injektive,
***    i1/=i2 ==> index(i1)/=index(i2)
***    compiler needs information about data dependency
***    faster with directive

*cdir  permutation(index)
*cdir nodep
      do 10 i=1,n
        a(index(i))=a(index(i))+b(i)
10    continue

      return
      end
```


gathering data

```
      subroutine gather_loop(a,index,n)
      real a(n)
      integer index(n)
***    gather
***    vectorizable

      j=0
      do 10 i=1,n
        if(a(i).ne.0.) then
          j=j+1
          index(j)=i
        endif
10    continue

      return
      end
```

multiple gather

```
      subroutine multiple_gather_loop(a,b,ind1,ind2,n)
      real a(n),b(n)
      integer ind1(n),ind2(n)
***   two gather
***   vectorizable
      j1=0
      j2=0
      do 10 i=1,n
        if(a(i)*b(i).gt.0.) then
          j1=j1+1
          ind1(j1)=i
        elseif(a(i).lt.0.) then
          j2=j2+1
          ind2(j2)=i
        endif
10    continue
      return
      end
```


scattering data

```
      subroutine scatter_loop(a,b,index,n)
      real a(n),b(n)
      integer index(n)
***   vectorizable even if index not unique

      do 10 i=1,n
      a(index(i))=b(i)
10    continue

      return
      end
```

vectorizable but not data parallel

small inner loops

```
      subroutine small_inner_loop(a,b,c,n)
      real a(n,3),b(n,3),c(n,3)
***    inner loop unrolled by compiler

      do 10 i=1,n
        do 20 m=1,3
          a(i,m)=b(i,m)+c(i,m)
20      continue
10    continue

      return
      end
```


small array in derived type

```
module vector_module
type vector_type
real    :: xx(3)
endtype vector_type
end module vector_module

subroutine add_vectors(aa,bb,n)
use vector_module
integer                 :: n
type(vector_type)  :: aa(n),bb(n)

do  i = 1,n
     aa(i)%xx = bb(i)%xx + aa(i)%xx
enddo
return
end
```

strided access

```
      subroutine strided_add(a,b,c,n)
      real a(16,n),b(16,n),c(16,n)
***! better: real a(16+1,n),b(16+1,n),c(16+1,n)
***  very slow for stride 16
***  slow also on cache machines

      do m=1,16
      do i=1,n
        a(m,i)=b(m,i)+c(m,i)
      enddo
      enddo

      return
      end
```


matrix diagonal

```
      subroutine matrix_diagonal(a,b,c,n)
      real a(n,n),b(n,n),c(n,n)
***   constant stride!    a(i,i) identical with a( i*(n+1)-n , 1 )
***   vectorizable

      do 10 i=1,n
      a(i,i)=b(i,i)+c(i,i)
10    continue

      return
      end
```

induction variable with conditional increment

```
      subroutine induction_variable_under_an_if(a,b,c,n)
      integer n
      real a(*),b(*),c(*)
*** not vectorizable

      j = 0
      do 50 i = 1,n
        j = j + 1
        a(j) = b(i)
        if(c(i).gt.0) then
          j = j + 1
          a(j) = c(i)
        endif
   50 continue
      return
      end
```


induction variable with unique increment

```
      subroutine induction_variable_under_ifelse(a,b,c,n)
*     induction variable under both sides of if (same value)
      integer n
      real a(*),b(*),c(*)
*** vectorizable
      j = 0
      do 60 i = 1,n
        if(b(i).gt.0) then
          j = j + 1
          a(j) = b(i)
          else
          j = j + 1
          a(j) = c(i)
        endif
   60 continue
      return
      end
```

transpose of triangular matrix with addition

```
      subroutine assign_lower_triangle_to_upper(aa,bb,n)
      integer n
      real aa(n,*),bb(n,*)
*** vectorizable

      do 300 j = 1,n
        do 300 i = 1,j-1
          aa( i , j ) = aa( j , i ) + bb( i , j )
300 continue
      return
      end
```


unrolling a long loop

```
subroutine unrolled_loop(a,b,x,n)
integer n
real a(*),b(*),x

 x=0.
do i = 1,n,4                  ! necessary rest of loop,
                              ! stride on vector machines !
   x = x + a(i    )*b(i    )
&        + a(i+1)*b(i+1)
&        + a(i+2)*b(i+2)
&        + a(i+3)*b(i+3)
     enddo
 return
 end
```

unrolling on cache machines

```
subroutine unrolled_loop_2(a,b,sum,n)
 ! for cache machines
integer n
real a(*),b(*),sum_1,sum_2,sum_3,sum_4

 sum_1=0. ; sum_2=0.; sum_3=0. sum_4=0.
do i = 1,n,4                                  ! rest of loop not included !
   sum_1 = sum_1 + a(i    )*b(i    )
   sum_2 = sum_2 + a(i+1)*b(i+1)
   sum_3 = sum_3 + a(i+2)*b(i+2)
   sum_4 = sum_4 + a(i+3)*b(i+3)
enddo
sum=sum_1+sum_2+sum_3+sum_4
return
end
```


intrinsics

```
      subroutine handling_intrinsics(a,b,c,n)
      integer n
      real a(*),b(*),c(*)
*** vectorizable

      do i = 1,n
        a(i) = sin(b(i)) + cos(c(i))
      enddo

      return
      end
```

working with strings

```
subroutine loop_with_character(a,b,n,error_type)
  integer n
  real a(*),b(*)
  character(len=*)    :: error_type
*** not vectorizable
      error_type=''
  do i = 1,n
    if( b(i) <= 0 ) then
      error_type='log for value less than 0!'
    else
      a(i) = log(b(i))
    endif
  enddo

  return
  end
```


replacing working with strings

```
subroutine loop_with_character(a,b,n,error_type)
  integer                    :: n
  real                       :: a(*),b(*)
  character(len=*)           :: error_type
*** vectorizable
      error_type=' ' ;   imark=0
  do i = 1,n
    if( b(i) <= 0 ) then
      imark=i
    else
      a(i) = log(b(i))
    endif
  enddo
  if( imark>0 ) error_type='log for value less than 0!'
  return
  end
```

linked list: derived type

```
type edge_pointer_type
real(kind=real_kind)                  :: var1
real(kind=real_kind)                  :: var2
type(edge_pointer_type),pointer       :: edge_1
type(edge_pointer_type),pointer       :: edge_2
end type edge_pointer_type
```


linked list: crossing a loop

```
subroutine linked_list(edge,list_length)
type(edge_pointer_type),pointer :: edge
type(edge_pointer_type),pointer :: first_edge

list_length=0
first_edge=>edge
edge => edge%edge_1
do
   edge%var1 =  edge%edge_1%var2+edge%edge_2%var2
   list_length=list_length+1
   if( associated(edge,first_edge) ) exit
   edge => edge%edge_1
enddo
end subroutine linked_list
```

note 12.21 in Fortran Standard

```
subroutine add(a,b,c,n)
    real,dimension(*)  :: a,b,c

                          !  could be equivalent to recursive loop
     do i=1,n             ! do i=1,m-1
       a(i)=b(i)+c(i)     !    x(i+1)=x(i)+y(i)
     enddo                ! enddo

    end subroutine add

   subroutine C_Fortran_difference(x,y,m)
    real, dimension(m) :: x,y

     call add(x(2),x(1),y,m-1) ! forbidden by Fortran Standard

    end subroutine C_Fortran_difference
```