Reporting and Listing Compiler Options

IA32 (GNU compilers)

None

IA32 (PGI compilers)

`-Minfo=option[,option,...]`	Prints information to `stderr` on `option`; **`option` can be one or more of `time`, `loop`, `inline`, `sym`, or `all`**
`-Mneginfo=option[,option]`	Prints information to `stderr` on why optimizations of type `option` were not performed; **`option` can be `concur` or `loop`**
`-Mlist`	Generates a listing file

IA64 (Intel compilers)

None

Floating Point Division

```
program fdiv
  integer,parameter :: N=10000000
  real,dimension(N):: a
  real:: begint,endt
  real,dimension(2):: rtime
  common/saver/a
    call random_number(a)
    x=3.71
    begint=dtime(rtime)
!DIR$ UNROLL 8
    do i=1,N
      a(i)=a(I) / x
    end do
    endt=dtime(rtime)
    print *,' loop time',endt
    sum = 0.
    do i=1,n
      sum = sum + a(i)
    end do
    print *, sum
end
```

FP Division Compiler Options

IA32 and IA64 (GNU compilers)

None

IA32 (PGI compilers)

`-Knoieee`	Use inline division and disable traps on underflow

IA64 (Intel compilers)

`-mp`	Maintain floating point precision by not transforming division into multiplication by a reciprocal

Compile fdiv with and without the flag to relax IEEE requirements (use –O0)

Was there any performance difference?

Why?

Floating Point Array Division

```
program arraydiv
  integer,parameter :: N=5000
  real,dimension(N,N):: a
  real,dimension(N):: b
  real:: begint,endt
  real,dimension(2):: rtime
  common/saver/a,b
    call random_number(a)
    call random_number(b)
    begint=dtime(rtime)
     do j=1,N
       do i=1,N
         a(i,j)=a(i,j)/b(i)
       end do
    end do
    endt=dtime(rtime)
    print *,' loop time',endt
    print *,a(N,N)
 end
```

```
program arraydiv2
  integer,parameter :: N=5000
  real,dimension(N,N):: a
  real,dimension(N):: b,tmp
  real:: begint,endt
  real,dimension(2):: rtime
  common/saver/a,b
    call random_number(a)
    call random_number(b)
    begint=dtime(rtime)
    do i=1,N
      tmp(i)=1.0/b(i)
    end do
    do j=1,N
!DIR$ UNROLL 8
      do i=1,N
        a(i,j)=a(i,j)*tmp(i)
      end do
    end do
    endt=dtime(rtime)
    print *,' loop time',endt
    print *,a(N,N)
end
```

Compile and run each program with –O2 –Minfo=loop

What was the difference in performance?

Try stripmining

Loop Interchange

```
program badstride
  integer,parameter :: N=1024,M=1024,ntimes=100
  real,dimension(N,M) :: a,b
  real,dimension(N) :: c
  real:: begint,endt
  real,dimension(2) :: rtime
  call random_number(b)
  call random_number(c)
  begint=dtime(rtime)
  do it=1,ntimes
    do i=1,M
      do j=1,N
        a(i,j)=b(i,j)+c(i)
      end do
      c(i) = c(i) + 0.5
    end do
    print *,a(N,M),b(N,M)
  end do
  endt=dtime(rtime)
  print *,‘ loop time:',endt
  flop=(ntimes*(N*M+M)/endt*1.0e-6
  print *,'loop runs at ',flop,' MFLOP'
 end
```

```
program goodstride
  integer,parameter :: N=1024,M=1024,ntimes=100
  real,dimension(N,M) :: a,b
  real,dimension(N) :: c
  real:: begint,endt
  real,dimension(2) :: rtime
  call random_number(b)
  call random_number(c)
  begint=dtime(rtime)
  do it=1,ntimes
    do j=1,M
      do i=1,N
        a(i,j)=b(i,j)+c(i)
      end do
      c(i) = c(i) + 0.5
    end do
    print *,a(N,M),b(N,M)
  end do
  endt=dtime(rtime)
  print *,‘ loop time:',endt
  flop=(ntimes*N*M+M)/endt*1.0e-6
  print *,'loop runs at ',flop,' MFLOP'
end
```

Compile and run each program with -O2 -Munroll -Minfo=loop

What was the difference in performance?

Compile badstride with -Munroll -Mvect -Minfo=loop bad.f90

What happened now?

Loop Interchange Compiler Options

IA32 and IA64 (GNU compilers)
None

IA32 (PGI compilers)
`-Mvect`
Enables vectorization, including loop interchange

IA64 (Intel compilers)
`-03`
Enables aggressive optimization, including loop transformations

Cache Trashing

```
program thrash
  integer,parameter :: N=4*1024*1024
  real,dimension(N) :: c,b,a,e,f,g
  real:: begint,endt
  real,dimension(2):: rtime
  common/saver/a,b,c,e,f,g
!DIR$ CACHE_ALIGN /saver/
  call random_number(b)
  call random_number(c)
  call random_number(e)
  call random_number(f)
  begint=dtime(rtime)
  do i=1,N
    a(i)=b(i)+1.5.*c(i)
    g(i)=e(i)+0.7.*f(i)
  end do
  endt=dtime(rtime)
  print *,' loop time: ',endt
  flop=4*N/endt*1.0e-6
  print *,‘ loop runs at ',flop,' MFLOP'
  print *,a(N),b(N),c(N)
 end
```

```
program pad
  integer,parameter :: N=4*1024*1024
  real,dimension(N) :: c,b,a,e,f,g
  real:: begint,endt
  real,dimension(2):: rtime
 common/saver/a,d0(4),b,d1(4),c,d2(4),e,d3(4),f,d4(4),g
!DIR$ CACHE_ALIGN /saver/
  call random_number(b)
  call random_number(c)
  call random_number(e)
  call random_number(f)
  begint=dtime(rtime)
  do i=1,N
    a(i)=b(i)+1.5*c(i)
    g(i)=e(i)+0.7.*f(I)
  end do
  endt=dtime(rtime)
  print *,' loop time: ',endt
  flop=4*N/endt*1.0e-6
  print *,‘ loop runs at ',flop,' MFLOP'
  print *,a(N),b(N),c(N)
end
```

Compile and run each program with –O2 –Munroll –Minfo=loop

What there any difference in performance?

Cache Line Alignment

IA32 and IA64 (GNU compilers)

`-malign-double`	Align double precision variables on 64-bit boundaries

IA32 (PGI compilers)

`-mcache_align`	Align arrays not static or in COMMON blocks on cache line boundaries
`-Mdalign`	Align doubles [i.e. REAL*8] in Fortran COMMON blocks and structures on 8-byte boundaries

IA64 (Intel compilers)

`-zp8`	Specify alignment constraint for structures on 8-byte boundaries; default

Internal Padding

```
program ctrash
  integer,parameter :: N=1024,M=64,ntimes=100
  real,dimension(N,N) :: a,b
  real:: begint,endt
  real,dimension(2):: rtime
  common/saver/a,b
!DIR$ CACHE_ALIGN /saver/
  call random_number(b)
  begint=dtime(rtime)
  do it=1,ntimes
   do j=1,N-3
     do i=1,N
      a(i,j)=b(i,j)-b(i,j+1)+b(i,j+2)-b(i,j,3)
     end do
   end do
   if (A(N,N) .lt. 0) print *,it,a(N,N)
  enddo
  endt=dtime(rtime)
  print *,' loop time: ',endt
  flop=(ntimes*3*N*(N-3))/endt*1.0e-6
  print *,'loop runs at ',flop,' MFLOP'
 end
```

```
program intpad
  integer,parameter :: N=1024,M=64,ntimes=100
  real,dimension(N+5,N) :: a,b
  real:: begint,endt
  real,dimension(2):: rtime
  common/saver/a,b
!DIR$ CACHE_ALIGN /saver/
  call random_number(b)
  begint=dtime(rtime)
  do it=1,ntimes
   do j=1,N-3
     do i=1,N
      a(i,j)= b(i,j)-b(i,j+1)+b(i,j+2)-b(i,j,3)
     end do
   end do
   if (A(N,N) .lt. 0) print *,it,a(N,N)
  enddo
  endt=dtime(rtime)
  print *,' loop time: ',endt
  flop=(ntimes*3*N*(N-3))/endt*1.0e-6
  print *,'loop runs at ',flop,' MFLOP'
 end
```

Compile and run each program with –O2 –Munroll –Minfo=loop

What was the difference in performance?

Prefetching

```
program sum
  integer,parameter :: N=50000,ntimes=10000
  real :: x(N),y(N)
  real:: begint,endt
  real:: rtime(2)
  common/saver/x,y
!DIR CACHE_ALIGN
  call random_number(x)
  call random_number(y)
  begint=dtime(rtime)
  do it=1,ntimes
    do i=1,N
      y(i)=x(i)+i
    end do
    if (mod(it,1000).eq.0) then
     print *,it,x(1),y(1)
     x(1) = x(1) – 0.25
  end do
  endt=dtime(rtime)
  print *,‘ loop time: ',endt
  flop=(N*ntimes)/endt*1.0e-6
  print *,' loop runs at ',flop,' MFLOP'
 end
```

Compile with: -O2 -Munroll -Minfo=loop

Then compile with -O2 -Munroll -Mvect=prefetch -Minfo=loop

Was there any difference in performance?

Prefetching Compiler Options

IA32 and IA64 (GNU compilers)

None

IA32 (PGI compilers)

`-Mvect=prefetch` — Enable prefetching

IA64 (Intel compilers)

`-O3` — Enable aggressive optimization, including prefetching

Inlining

```
program inline
  parameter (N=5000000)
  real a,x,y,z
  real begint,endt,rtime
  a=28.456890
  begint=dtime(rtime)
  do i=1,N
    call hyp(a,x,y,z)
  end do
  endt=dtime(rtime)
  print*,' loop time: ',endt
  print*,x,y,z
end
subroutine hyp(a,x,y,z)
  real a,x,y,z
  call random_number(x)
  call random_number(y)
  z=sqrt(y**2+a*x**2)
  return
end
```

Compile with: -O2 -Munroll -Minfo=loop

Did it unroll the loop?

Then compile with -O2 -Minline inline.f90 -o inline.ia32

Was there any difference in performance?

Inlining Compiler Options

IA32 and IA64 (GNU compilers)

-fno-inline	Disable inlining
-finline-functions	Enable inlining of functions

IA32 (PGI compilers)

-Mextract=option[,option,…]	Extract functions selected by **option** for use in inlining; **option** may be **name:function** or **size:N** where **N** is a number of statements
-Minline=option[,option,…]	Perform inlining using **option**; **option** may be **lib:filename.ext**, **name:function**, **size:N**, or **levels:P**

IA64 (Intel compilers)

-ip	Enable single-file interprocedural optimization, including enhanced inlining
-ipo	Enable interprocedural optimization across files