EFFECTIVE PARALLEL OPTIMIZATIONS
WITH INTEL® FORTRAN

Martyn Corden
Intel Corporation
March 2016
Agenda

SIMD parallelism
- Auto-vectorization
- Explicit SIMD programming with OpenMP* 4.5

Multi-core parallelism
- Auto-parallelism
- OpenMP

Important, but not part of the compiler and not addressed to day:
- Intel® Math Kernel Library
- MPI

For brevity, most examples will use the compiler switch syntax for Linux* and OS* X. Equivalent syntax for Windows* is documented in the Intel® Compiler User and Reference Guide.
SIMD: Single Instruction, Multiple Data

Scalar mode
- one instruction, one result
- E.g. vaddss, (vaddsd)

Vector mode
- with Intel® AVX (or SSE) instructions
- one instruction, multiple results
- E.g. vaddps, (vaddpd)

![Diagram showing SIMD operations in scalar and vector modes]
Agenda

SIMD parallelism
  • Auto-vectorization
  • Explicit SIMD programming with OpenMP* 4.5

Multi-core parallelism
  • Auto-parallelism
  • OpenMP
Factors that have Impact on Auto-Vectorization

Loop-carried dependencies

```
DO I = 1, N
  A(I+IOFFSET) = A(I) + B(I)
ENDDO
```

Function or subroutine calls

```
DO I = 1, N
  SUMX = SUMX + FUNC(X(I), X0)
ENDDO
```

Pointer aliasing

```
REAL, POINTER, DIMENSION(:) :: A, B, C
DO I = 1, N
  A(I) = B(I) + 2.*C(I)
ENDDO
```

Unknown iteration count / multiple exits

```
DO I = 1, N
  IF(A(I) < 0.) EXIT
  A(I) = SQRT(A(I))
ENDDO
```

Indirect memory access / non-unit stride

```
DO I = 1, N
  A(I) = B(I)*C(INDEX(I))
ENDDO
```

OK: IF statements, math functions, scalar assignments

```
DO I=1,N
  S = B(I)**2 - 4.*A(I)*C(I)
  IF ( S .GT. 0 ) X(I) = (-B(I)+SQRT(S))/(2.*A(I))
ENDDO
```

many more ......
Guidelines for Writing Vectorizable Code

Prefer simple “DO” or “for” loops

Write straight line code. Avoid:
• Most function or subroutine calls
• Branches that can’t be treated as masked assignments.

Avoid dependencies between loop iterations
• Or at least, avoid read-after-write dependencies

Prefer arrays to the use of pointers or “associate”
• The compiler often cannot tell whether it is safe to vectorize code containing pointers.
• Try to use the loop index directly in array subscripts, instead of incrementing a separate counter for use as an array address.

Use efficient memory accesses
• Favor inner loops with unit stride
• Minimize indirect addressing
• Align your data consistently where possible e.g. -align array64byte
More General Advice

Avoid manual unrolling in source (common in legacy codes)
  • (re)write as simple DO or for loops
  • Easier for the compiler to optimize and align
  • Less platform-dependent
  • More readable

Make loop induction variables local (including loop limits)
  • Compiler knows they can’t be aliased

Beware Fortran pointer and assumed shape array arguments
  • Compiler can’t assume they are unit stride
    • Declare CONTIGUOUS where appropriate
  • Prefer allocatable arrays to pointers where possible
Assumed Shape Array Dummy Arguments ...

module mod2
contains
  subroutine sub2(a,b,c)
    real, dimension(:) :: a,b,c
    a = b + abs(c)
  end subroutine sub2
end module mod2

module mod1
contains
  subroutine sub1(a,b,c)
    use mod2
    real, dimension(:) :: a,b,c
    call sub2(a,b,c)
  end subroutine sub1
end module mod1

ifort -c -qopt-report=4 sub2.f90

... LOOP BEGIN at sub2.f90(6,5)
remark #15335: loop was not vectorized: vectorization possible but seems inefficient. ...
remark #15329: vectorization support: scatter was emulated for the variable a: strided by non-constant value
remark #15328: vectorization support: gather was emulated for the variable b: strided by non-constant value
remark #15328: vectorization support: gather was emulated for the variable c: strided by non-constant value

ifort -c -qopt-report=4 sub2.f90

(no messages)
... could be Array Sections

Compiler must be conservative, assume arguments may have non-unit stride

- e.g.  CALL SUB2 ( A(1:1000), B(1:2000:2), C(1:4000:4) )
  - Compiler does not generate SIMD (vector) loads (→ ‘gather’ of individual loads)
  - Makes vectorization not worthwhile in this example
    - Loops containing more work may still vectorize, but run slower

Solution: If arrays have unit stride, tell the compiler!

- REAL, DIMENSION(:), CONTIGUOUS :: A,B,C        (new in Fortran 2008)

  remark #15300: LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 1

Alternatively, use assumed size or variable length arrays, e.g.
  Subroutine sub2(a, b, c, n)
  real, dimension(n) :: a, b, c
Impact of **CONTIGUOUS** on Caller

But look what happens to sub1, now that interface to sub2 has changed!

- Many vectorization messages, like

  LOOP BEGIN at sub1.f90(7,10)
  <Multiversioned v1>
  remark #15300: LOOP WAS VECTORIZED
  LOOP END

  LOOP BEGIN at sub1.f90(7,10)
  <Multiversioned v2>
  remark #15335: loop was not vectorized: vectorization possible but seems inefficient. ...
  LOOP END

- Even though the source code contains no loops!
- Compiler is making a contiguous copy of each argument, as required for sub2
  - We just pushed the problem back upstream!
  - Need to declare arrays as **CONTIGUOUS** back up the call stack
  - Then the loops in sub1 go away
Module mod2
contains
  subroutine sub2(a,b,c)
    real, dimension(:), contiguous :: a,b,c
    a = b + abs(c)
  end subroutine sub2
end module mod2

ifort -c -qopt-report=4 sub2.f90

... LOOP BEGIN at sub2.f90(6,5)
  remark #15300: LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 1
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15450: unmasked unaligned unit stride loads: 1

Module mod1
contains
  subroutine sub1(a,b,c)
    use mod2
    real, dimension(:), contiguous :: a,b,c
    call sub2(a,b,c)
  end subroutine sub1
end module mod1

ifort -c -qopt-report=4 sub2.f90
  (no messages)
# Performance Impact

(array size 1000, executed 2000000 times)

<table>
<thead>
<tr>
<th>Sub1 contiguous</th>
<th>Sub2 contiguous</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>1.0</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>0.85</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>1.0</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary.

The results above were obtained on a 4th Generation Intel® Core™ i7-4790 system, frequency 3.6 GHz, running Red Hat® Enterprise Linux® version 7.0 and using the Intel® Fortran Compiler version 16.0.2.
Agenda

SIMD parallelism

• Auto-vectorization
• Explicit SIMD programming with OpenMP* 4.5

Multi-core parallelism

• Auto-parallelism
• OpenMP
Explicit Vector Programming with OpenMP*

Vectorization is so important ➔ consider explicit vector programming

Modeled on OpenMP* for threading (explicit parallel programming)

• Enables reliable vectorization of complex loops that the compiler can’t auto-vectorize
  • E.g. outer loops (see webinar from 1 year ago: https://software.intel.com/videos/new-vectorization-features-of-the-intel-compiler

• Directives are commands to the compiler, not hints
  • Programmer is responsible for correctness (like OpenMP threading)
    • E.g. PRIVATE, FIRSTPRIVATE and REDUCTION clauses

• Now incorporated in OpenMP 4.0 ⇒ portable
  • -qopenmp or -qopenmp-simd to enable
The !$OMP SIMD directive

Enabled by -qopenmp or -qopenmp-simd ( /Qopenmp or /Qopenmp-simd )

```
subroutine add(A, N, X)
    integer N, X
    real A(N)
    !$OMP SIMD
    do i=x+1, n
        a(i) = a(i) + a(i-x)
    enddo
end
```

ifort -c -qopt-report-file=stderr add.f90
...
loop was not vectorized: existence of vector dependence.

ifort -c -qopt-report-file=stderr -qopenmp-simd add.f90
...
SIMD LOOP WAS VECTORIZED

Use when you KNOW that a given loop is safe to vectorize

The Intel® Compiler will vectorize if at all possible (ignoring dependency or efficiency concerns)

https://software.intel.com/en-us/articles/requirements-for-vectorizing-loops-with-pragmasimd

Minimizes source code changes needed to enforce vectorization
Loops Containing Subroutine Calls

Function calls can have side effects that introduce a loop-carried dependency, preventing auto-vectorization. Possible remedies:

• Inlining
  • best for small procedures
  • Must be in same source file, or else use -ipo (/Qipo on Windows*)

• SIMD-enabled functions or subroutines
  • Good for large, complex routines and in contexts where inlining is difficult
  • Call from regular DO loop
  • Adding “ELEMENTAL” keyword allows a SIMD-enabled procedure to be called with an array section argument

• !$OMP SIMD directive (last resort)
  • to vectorize remainder of loop, while preserving scalar calls to procedure
SIMD-enabled Subroutine

Compiler generates SIMD-enabled (vector) version of a scalar subroutine that can be called from a vectorized loop:

```fortran
subroutine test_linear(x, y)
  !$omp declare simd (test_linear) linear(ref(x, y))
  real(8), intent(in) :: x
  real(8), intent(out) :: y
  y = 1. + sin(x)**3
end subroutine
```

```
do j = 1,n
  call test_linear(a(j), b(j))
endo
```

```fortran
ifort -xavx -qopenmp-simd -qopt-report-file=stderr test_linear.f90
```

SIMD-enabled routine must have `explicit interface` ( !$omp simd not needed in simple cases like this )
SIMD-enabled Subroutine - Performance

The **LINEAR(REF())** clause is very important (New in compiler version 16.0.1)

- In C, compiler places consecutive argument values in a vector register
- But Fortran passes arguments by reference
  - By default compiler places consecutive addresses in a vector register
  - Leads to a gather of the 4 addresses (slow)
  - **LINEAR(REF(X))** tells the compiler that the addresses are consecutive; only need to dereference once and copy consecutive values to vector register
- Same method could be used for C arguments passed by reference

### Approx speed-up for double precision array of 1M elements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No DECLARE SIMD</td>
<td>1.0</td>
</tr>
<tr>
<td>DECLARE SIMD but no LINEAR(REF)</td>
<td>0.9</td>
</tr>
<tr>
<td>DECLARE SIMD with LINEAR(REF) clause</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Performance tests are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. The results above were obtained on an Intel® Xeon® E7-4850 v3 system, frequency 2.2 GHz, running Red Hat* Enterprise Linux* version 7.1 and using the Intel® Fortran Compiler version 16.0.1.
Agenda

SIMD parallelism
  • Auto-vectorization
  • Explicit SIMD programming with OpenMP* 4.5

Multi-core parallelism
  • Auto-parallelization
  • OpenMP
Multi-core Parallelism – Thread Safety

Applies to both OpenMP* and auto-parallelism
  • But not to coarrays

Compile you whole application with -qopenmp  (or -threads -auto)
  • Including “orphaned” subroutines containing no OpenMP directives
    • helps make local arrays thread safe
      • -parallel alone is not sufficient for this
      • often requires increased stack size
  • Including the main routine
    • Helps link and initialize the Fortran RTL in threadsafe mode

To make routines thread safe in source code (independent of command line)
  • Declare function as RECURSIVE
  • Don’t use -save or the SAVE keyword
  • Avoid global variables (or don’t write to them unless synchronized)
Auto-parallelization (auto-threading)

Works very like auto-vectorization

- -parallel (/Qparallel) to enable
- Makes use of OpenMP* run-time library
- Iterations must be completely independent of each other
- Threading overhead is >>> vectorization overhead (order ~$10^4$ cycles)
  - Only worthwhile for loops with a lot of computational work
  - Function calls need to be inlined (using ipo, if necessary)

Eventually, DO CONCURRENT (Fortran 2008) will help

- Asserts loop is safe for parallel execution
- Any called subroutines or functions must be PURE (no side effects)
- So must have explicit interface
- Must be threadsafe (-auto or recursive; -parallel is not sufficient)
- Not yet fully implemented for function calls 😞 - wait for fix to dpd200378091
DO CONCURRENT example

ifort -c -parallel test.f90 -qopt-report-file=stderr
...
LOOP BEGIN at test.f90(15,3)
  remark #17104: loop was not parallelized: existence of parallel dependence
  remark #15344: loop was not vectorized: vector dependence prevents vectorization.
Replace “DO” by “DO CONCURRENT”:

ifort -c -parallel test.f90 -qopt-report-file=stderr
...
LOOP BEGIN at test.f90(16,3)
  remark #17109: LOOP WAS AUTO-PARALLELIZED
  remark #15527: loop was not vectorized: function call to sub cannot be vectorized [ test.f90(19,12) ]
LOOP END

pure recursive subroutine sub (y, z)
  real, intent(in) :: y
  real, intent(out) :: z
  z = 1. + sin(y)**2
end subroutine
...
do i=1, n
! do concurrent (i=1:n)
  call sub(b(i), c(i))
enddo
Agenda

SIMD parallelism
  • Auto-vectorization
  • Explicit SIMD programming with OpenMP* 4.5

Multi-core parallelism
  • Auto-parallelism
  • OpenMP
Using OpenMP* with Intel® Fortran

Not teaching the OpenMP* standard
• See www.openmp.org

Straightforward, but programmer is responsible for correctness
• Private, reduction, etc. clauses

Start with optimized serial code
• vectorized inner loops
• -O3 -xcore-avx2 -ipo ...

Thread outermost loops where possible

Same loop can be threaded and vectorized
• Where warranted by the amount of work
• Auto- or explicit vectorization
• $omp parallel do simd ....

```fortran
!$omp parallel do private(a) &
!$omp reduction(+:sum)
do i=1, n
   a = real(i)
   a = 1. + a + a**2
   call sub(a, b(i), c(i))
   sum = sum + a
endo
```
Performance considerations

Avoid false sharing of cache lines
- Each thread thinks it's copy of A(i,j) may have been invalidated
- Reversing subscripts of A improves data locality for each thread
  - Contiguous memory access also permits vectorization of inner loop

Scheduling options
- Consider DYNAMIC or GUIDED if work is unevenly distributed between loop iterations

Set thread/processor affinity with OMP_PROC_BIND
- “spread” works well for most applications, makes use of all cores & sockets
- “close” places threads as close together as possible, maximum threads per core
  - maximizes use of shared cache, but might leave some cores or sockets idle
- Default is no affinity - not usually what you want
- KMP_AFFINITY (Intel-specific) gives finer control

```c
!$OMP parallel do
do i=1,nthreads
do j=1,10000
  A(i,j) = A(i,j) + ..
```
Thread at a High Level

More work in parallel region -> better scaling;

Makes thread safety easier:

!$omp parallel do private(a,b,c,d,e,f,g,h,i1,i2 ...)
Do  i = 1, n
    ....
!$omp parallel do private(....)
Do  i = 1, n
    ....
becomes
!$omp parallel call sub()
    ...
Easier to maintain!

subroutine sub
Real  :: a,b,c,d,e,f,
Integer :: i1, i2
Real, dimension(ndim) :: g,h
! Local declarations are automatically threadsafe
! when compiled with -qopenmp
...
!$omp do
Do  i = 1, n
...
Common Problem – Insufficient Stacksize

Most frequently reported OpenMP issue!
- Typical symptom: seg fault during data initialization (due to stack overflow)
- -qopenmp (or -auto) causes local arrays to be allocated on stack for thread safety

Shell (master thread) stack limit:
- Applies to whole program, shared + local data (whether or not threaded)
- typical OS defaults are small, ~10MB
- It's an address space limit; memory is allocated only if needed
  - Safe to set to large value
  - Bash: ulimit –s [value in KB] or [unlimited] (can only increase once!)
  - C shell: limit stacksize [value in KB]
  - Windows*: link with /F:100000000 (value in bytes)

Individual thread stack allocation (thread local data only):
- export OMP_STACKSIZE=[size], default 4m (4 MB)
- Actually allocates memory, so don’t make too large
Debugging Apps threaded with OpenMP*

Debug at -O0. Unlike SIMD vectorization, threading is not disabled at -O0

Build with -qopenmp but run with a single thread (OMP_NUM_THREADS=1)
  • If it now works, try Intel® Inspector to detect race conditions, etc.

Build with -qopenmp-stubs -auto
  • RTL calls are resolved; no threaded code is generated
  • If it now works, check for missing FIRSTPRIVATE, LASTPRIVATE

Build with -qopenmp-stubs
  • If it works, implicates changed memory model with OpenMP*
    • Perhaps insufficient stacksize (see previous slide)
    • Perhaps values not preserved between successive procedure calls

If debugging with PRINT statements
  • USE OMP_LIB before printing thread number with omp_get_thread_num()
  • Internal I/O buffers are threadsafe
  • Order of print statements from different threads is undetermined
One Word on Coarrays  (from Fortran 2008)

Advantages:
- Same code for shared and distributed memory parallelism
- Naturally thread safe
- Easier to learn than MPI

Disadvantages
- Still need to set up cluster environment
  - Passwordless ssh, etc.
- MPI is (far) more feature rich
- Communication overhead is (much) worse than for MPI
  - Substantial improvements in 17.0 compiler

Still need plenty of work to amortize overhead

```
real, dimension(n), codimension[*] :: b,c

nimages = num_images()
myid   = this_image()
print *, this_image =', myid, ' /', nimages
! Initialize data first
...
  do i = myid, n, nimages
    a = real(i)
    a = 1. + sin(a)**2
    call sub(a, b(i), c(i))
  enddo
  sync all ! make results available everywhere
```

`ifort -coarray test_ca.f90; ./a.out`
For More Information...

Webinars:

Explicit Vector Programming in Fortran:
https://software.intel.com/articles/explicit-vector-programming-in-fortran

Initially written for Intel® Xeon Phi™ coprocessors, but also applicable elsewhere:
https://software.intel.com/articles/vectorization-essential

Intel® Compiler User and Reference Guide:

