Vectorizing Fortran using OpenMP 4.x
Filling the SIMD Lanes

Ronald W Green
May 2015
SIMD for Performance
Compiler Auto-Vectorization

SIMD – Single Instruction Multiple Data

- **Scalar mode**
  - one instruction produces one result

- **SIMD processing**
  - with SSE or AVX instructions
  - one instruction can produce multiple results

\[ c(\cdot) = a(\cdot) + b(\cdot) \]

\[
\text{do } i=1,\text{MAX} \\
\quad c(i) = a(i) + b(i) \\
\text{end do}
\]
Vectorization is Achieved through SIMD Instructions & Hardware

**Intel® SSE**
- Vector size: 128bit
- Data types:
  - 8, 16, 32, 64 bit integers
  - 32 and 64bit floats
- VL: 2, 4, 8, 16
- Sample: Xi, Yi bit 32 int / float

**Intel® AVX**
- Vector size: 256bit
- Data types: 32 and 64 bit floats
- VL: 4, 8, 16
- Sample: Xi, Yi 32 bit int or float
- First introduced in 2011
Vectorization is Achieved through SIMD Instructions & Hardware

**Intel® AVX-512**
Vector size: 512bit
Data types: 32 and 64 bit floats (other types vary by subsets)
VL: 4, 8, 16, 32
Sample: Xi, Yi 32 bit int or float
Announced in 2014

Terminology:
"LANE" is a parallel slice of elements used in a vector operation.
Think of this as similar to a multi-lane freeway

32bit operands

<table>
<thead>
<tr>
<th>512</th>
<th>255</th>
<th>128</th>
<th>127</th>
</tr>
</thead>
<tbody>
<tr>
<td>X16</td>
<td>X15</td>
<td>X14</td>
<td>X13</td>
</tr>
<tr>
<td>X12</td>
<td>X11</td>
<td>X10</td>
<td>X9</td>
</tr>
<tr>
<td>X8</td>
<td>X7</td>
<td>X6</td>
<td>X5</td>
</tr>
<tr>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
</tr>
<tr>
<td>Y16</td>
<td>Y15</td>
<td>Y14</td>
<td>Y13</td>
</tr>
<tr>
<td>Y12</td>
<td>Y11</td>
<td>Y10</td>
<td>Y9</td>
</tr>
<tr>
<td>Y8</td>
<td>Y7</td>
<td>Y6</td>
<td>Y5</td>
</tr>
<tr>
<td>Y4</td>
<td>Y3</td>
<td>Y2</td>
<td>Y1</td>
</tr>
<tr>
<td>Z16</td>
<td>Z15</td>
<td>Z14</td>
<td>Z13</td>
</tr>
<tr>
<td>Z12</td>
<td>Z11</td>
<td>Z10</td>
<td>Z9</td>
</tr>
<tr>
<td>Z8</td>
<td>Z7</td>
<td>Z6</td>
<td>Z5</td>
</tr>
<tr>
<td>Z4</td>
<td>Z3</td>
<td>Z2</td>
<td>Z1</td>
</tr>
</tbody>
</table>

Copyright © 2015, Intel Corporation. All rights reserved. *Other names and brands may be claimed as the property of others.
Auto-Vectorization – Options

{L&M} -x<extension> {W}: /Qx<extension>

Targeting Intel® processors - specific optimizations for Intel® processors

Compiler will try to make use of all instruction set extensions up to and including <extension>; for Intel® processors only!

Processor-check added to main-program

Application will not start (will display message), in case feature is not available

{L&M}: -m<extension> {W}: /arch:<extension>

No Intel processor check

Does not perform Intel-specific optimizations

Application is optimized for and will run on both Intel and non-Intel processors

Missing check can cause application to fail if instructions not supported

{L&M}:  -ax<extension> {W}: /Qax<extension>

Multiple code paths – a ‘baseline’ and ‘optimized, processor-specific’ path(s)

Optimized code path for Intel® processors defined by <extension>

Baseline code path defaults to –msse2 (Windows: /arch:sse2)

- The baseline code path can be modified by –m or –x (/Qx or /arch) switches
- axavx,sse4.2 - paths for avx, sse4.2 and sse2 (default)
Basic Vectorization – Switches

Special switch -xHost (Windows: /QxHost)
Compiler checks compilation host processor and makes use of ‘latest’ instruction set extension available for THAT processor

The Xeon default is –msse2 (Windows: /arch:sse2)
Activated implicitly for –O2 or higher
Implies the need for a target processor with Intel® SSE2

Multiple Code Paths
Multiple extensions can be used in combinations like
-ax<ext1>,<ext2> (Windows: /Qax<ext1>,<ext2> )
-axavx,sse4.2,sse3 for example, OR
-axavx –xsse4.2 (AVX path, SSE4.2 path, default SSE2 path)
# Compiler Based Vectorization

## Extension Specification – 128 and 256 bit Vector Extensions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Streaming SIMD Extensions 2 (Intel® SSE2) as available in initial Pentium® 4 or compatible non-Intel processors</td>
<td>sse2</td>
</tr>
<tr>
<td>Intel® Streaming SIMD Extensions 3 (Intel® SSE3) as available in Pentium® 4 or compatible non-Intel processors</td>
<td>sse3</td>
</tr>
<tr>
<td>Supplemental Streaming SIMD Extensions 3 (SSSE3) as available in Intel® Core™2 Duo processors</td>
<td>ssse3</td>
</tr>
<tr>
<td>Intel® SSE4.1 as first introduced in Intel® 45nm Hi-K next generation Intel Core™ micro-architecture</td>
<td>sse4.1</td>
</tr>
<tr>
<td>Intel® SSE4.2 Accelerated String and Text Processing instructions supported first by Intel® Core™ i7 processors</td>
<td>sse4.2</td>
</tr>
<tr>
<td>Extensions offered by Intel® ATOM™ processor : Intel SSSE3 (!!) and MOVBE instruction</td>
<td>sse3_atom</td>
</tr>
<tr>
<td>Intel® Advanced Vector Extensions (Intel® AVX) as available in 2nd generation Intel Core processor family – code name Sandy Bridge</td>
<td>AVX</td>
</tr>
<tr>
<td>Intel® Advanced Vector Extensions (Intel® AVX) as code-name Ivy Bridge and in code-name Haswell (available only in compilers v13+, fall 2012)</td>
<td>CORE-AVX-I CORE-AVX2</td>
</tr>
</tbody>
</table>
Compiler Based Vectorization

*Extension Specification – 512bit AVX-512 Vector used with –x<extension> or –ax<extension>*

**COMMON-AVX512**
Common to future Intel® Architecture Processors and future Intel® MIC Architecture code name Knights Landing

**MIC-AVX512**
Specific to future Intel® MIC Architecture code name Knights Landing – common PLUS prefetch and FP Exp and recip instructions

**CORE-AVX512**
Specific to future Intel® Architecture Processors: common plus more integer, byte, word instructions

May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.


May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation instructions, Intel® AVX-512 Conflict Detection instructions, Intel® AVX-512 Doubleword and Quadword instructions, Intel® AVX-512 Byte and Word instructions and Intel® AVX-512 Vector Length extensions, as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.
Other Optimizations that Support or Complement Vectorization

Vectorization relies on

- NO Dependencies in data between loop iterations
  - or dependencies that can be removed by simple methods

- Regular, predictable data patterns for all operands
  - No pointer chasing, indirect accesses

- Vector lengths that are large enough AND

- Data is aligned on natural boundaries (16, 32, or 64 bytes)
  - cache line matching boundary, can use cache as staging

- Streaming stores – store optimization

COST MODEL – is it worth it to vectorize???

Advanced optimizations help with some of these
Optimization Report for Vectorization

Understanding Which Loops Are Vectorized
Optimization Report

“Loop was not vectorized” because:

- “Low trip count”
- “Not Inner Loop”
- “Existence of vector dependence”
- "vectorization possible but seems inefficient"
- “nonstandard loop is not a vectorization candidate”
- “data type unsupported”
- “Unsupported reduction”
- “Condition may protect exception”
- “Statement cannot be vectorized”
- “Subscript too complex”
- ‘Unsupported Loop Structure”
- “Top test could not be found”
- “Operator unsuited for vectorization”
- ... (some more)

For more detail:


General

Applicable to Intel® Compiler version 15.0 +
for Fortran (and C/C++)

• for Windows*, Linux* and OS X*

  (For readability, options may not be repeated for each OS where spellings are similar. Options apply to all three OS unless otherwise stated.)

Main options:

-qopt-report[=N] (Linux and OS X)
/Qopt-report[:N] (Windows)

  N = 1-5 for increasing levels of detail, (default N=2)

-qopt-report-phase=str[,str1,...]

  str = loop, par, vec, openmp, ipo, pgo, cg, offload, tcollect, all

-qopt-report-file=[stdout | stderr | filename]
Report Output

Output goes to a text file by default, no longer stderr

- File extension is .optrpt, root is same as object file
- One report file per object file, in object directory
- created from scratch or overwritten (no appending)

-qopt-report-file=stderr gives old behavior (to stderr)

=filename to change default file name

/Qopt-report-format:vs format suitable for Visual Studio* IDE

For debug builds, (-g on Linux* or OS X*, /Zi on Windows*),
assembly code and object files contain loop optimization info

/Qopt-report-embed to enable this for non-debug builds
Loop, Vectorization and Parallelization Phases

Hierarchical display of loop nest

• Easier to read and understand

• For loops for which the compiler generates multiple versions, each version gets its own set of messages

Where code has been inlined, caller/callee info available

The “Loop” (formerly hlo) phase includes messages about memory and cache optimizations, such as blocking, unrolling and prefetching

• Now integrated with vectorization & parallelization reports
Loop Optimization Report

1 program matrix
2 !...a simple matrix multiply example
3 use iso_fortran_env
4 implicit none
5 integer, parameter :: sp=REAL32
6 integer, parameter :: dp=REAL64
7 integer, parameter :: ROWS=1000,COLS=1000,N=1000 ! square matrix example
8 real (kind=dp) :: a(ROWS,COLS)=2.0_dp, b(ROWS,COLS)=3.0_dp, c(ROWS,COLS)
9 integer :: i, j, k
10
11 c = 0.0_dp
12 do j=1,COLS
13 do i=1,ROWS
14 do k=1,N
15 c(i,j)=c(i,j)+a(i,k)*b(k,j)
16 end do
17 end do
18 end do
19 end program matrix

Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
...
Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
...
remark #15301: PERMUTED LOOP WAS VECTORIZED
...
Vectorization – report levels

-qopt-report-phase=vec  -qopt-report=N

N specifies the level of detail; default N=2 if N omitted
Level 0: No vectorization report
Level 1: Reports when vectorization has occurred.
Level 2: Adds diagnostics why vectorization did not occur.
Level 3: Adds vectorization loop summary diagnostics.
Level 4: Additional detail, e.g. on data alignment
Level 5: Adds detailed data dependency information

References for Vectorization Messages


The following diagnostic messages from the vectorization report produced by the Intel® Fortran Compiler. To obtain a vectorization report, use the -vec-report[n] option in Intel® Fortran for Linux® and Intel® Fortran Mac OS® X, or from the Intel® Visual Fortran for Windows /Qvec-report[n] option.

<table>
<thead>
<tr>
<th>Diagnostics Number</th>
<th>Diagnostic Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostic 15043</td>
<td>loop was not vectorized: nonstandard loop is not supported</td>
</tr>
<tr>
<td>Diagnostic 15038</td>
<td>loop was not vectorized: conditional assignment does not support the assignment of a scalar</td>
</tr>
<tr>
<td>Diagnostic 15015</td>
<td>loop was not vectorized: unsupported data type</td>
</tr>
<tr>
<td>Diagnostic 15011</td>
<td>loop was not vectorized: statement cannot be converted to a loop</td>
</tr>
<tr>
<td>Diagnostic 15046</td>
<td>loop was not vectorized: existence of vector does not support the assignment of a scalar</td>
</tr>
<tr>
<td>Diagnostic 15018</td>
<td>loop was not vectorized: not inner loop</td>
</tr>
<tr>
<td>Diagnostic 15002</td>
<td>LOOP WAS VECTORIZED</td>
</tr>
<tr>
<td>Diagnostic 15003</td>
<td>PARTIAL LOOP WAS VECTORIZED, OpenMP SIMD LOOP WAS VECTORIZED, REMAINDER LOOP WAS VECTORIZED, FUSED LOOP WAS VECTORIZED, REVERSED LOOP WAS VECTORIZED</td>
</tr>
</tbody>
</table>

Causes:

1. A loop contains a conditional statement
2. The conditional statement is controlling the assignment of a scalar value.
3. The logic of the assignment is such that the value of the scalar at the end of execution of the loop depends on the loop executing iterations strictly in-order AND
4. The scalar value is referenced AFTER the loop exits.

Below is an example:

```
01   subroutine f15038( a, b, n )
02       implicit none
03       integer :: a(n), b(n), n
04       integer :: i, x=10
05       !dir$ simd
06       do i=1,n
07           if( a(i) > 0 ) then
08               x = 1     !...here is the conditional assignment
09           end if
10           b(i) = x
11       end do
12       ! reference the scalar outside of the loop
13       write(*,*) "last value of x: ", x
14   end subroutine f15038
```
Assisting the Compiler with Vectorization

What Can I Do in My Programs to Assist the Compiler with Vectorization?
Guidelines for Writing Vectorizable Code: Fortran

**Prefer simple “DO” loops**

**Write straight line code. Avoid:**
- most function calls
- branches that can't be treated as masked assignments.

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

**Prefer allocatable arrays to pointers for dynamic memory allocation**
- 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,
- Annotate arguments with CONTIGUOUS attribute where applicable
- Minimize indirect addressing
- Try to align your data consistently where possible
  - to 32 byte boundaries (for AVX instructions) or 16 byte (SSE)
Intel Compiler Provided Directives

!DIR$ directives

- **IVDEP**: ignore vector dependency
- **LOOP COUNT**: advise typical iteration count(s)
- **UNROLL**: suggest loop unroll factor
- **DISTRIBUTE POINT**: advise where to split loop
- **VECTOR**: vectorization hints
  - **Aligned**: assume data is aligned
  - **Always**: override cost model
  - **Nontemporal**: advise use of streaming stores
- **NOVECTOR**: do not vectorize
- **NOFUSION**: do not fuse loops
- **INLINE/FORCEINLINE**: invite/require function inlining
- **SIMD**: explicit vector programming (see later)

Use where needed to help the compiler. Remember, these are Intel-compiler specific! We recommend the newer OpenMP 4 SIMD directives for many of these.
Essential Step #1: Align Data

Align the data at allocation ...

-align array64byte  compiler option to align all array types

Works for dynamic, automatic and static arrays (not in common)

Align COMMON array on an “n”-byte boundary (n must be a power of 2)

!dir$ attributes align:n :: array

And tell the compiler WHERE you use the data ...

!dir$ vector aligned

Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor

• May cause fault if data are not aligned

  !dir$ assume_aligned array:n [,array2:n2, ...]
  or !$omp SIMD aligned(list[:n])

• Compiler may assume (list) arrays are aligned to n byte boundary

n=64 MIC & AVX-512, n=32 for AVX, n=16 for SSE
A Quick Example on Unit Stride:

subroutine func( theta, sth )
implicit none
real :: theta(:,), sth(:,)!..can be contiguous or strided
integer :: i
do i=1,128
  sth(i) = sth(i) + (3.1415927D0 * theta(i))
end do
end
Actionable Messages, Fortran, Step 1

Begin optimization report for: FUNC

LOOP BEGIN at func_step1.f90(8,36)

<Peeled, Multiversioned v1>

LOOP END

LOOP BEGIN at func_step1.f90(8,36)

<Multiversioned v1>

remark #25233: Loop multiversioned for stride tests on Assumed shape arrays

remark #15388: vectorization support: reference sth has aligned access  [ func_step1.f90(8,3) ]

remark #15388: vectorization support: reference theta has aligned access  [ func_step1.f90(8,3) ]

<snip>

LOOP END

LOOP BEGIN at func_step1.f90(8,36)

<Alternate Alignment Vectorized Loop, Multiversioned v1>

remark #25015: Estimate of max trip count of loop=16

LOOP END

LOOP BEGIN at func_step1.f90(8,36)

<Remainder, Multiversioned v1>

==========================================================================

25 Actionable Messages, Fortran, Step 1

Loop multiversioned due to Assumed Shape arrays

One version assumes contiguous data. This version has PEEL + Kernel + Remainder loops

Another version assumes non-contiguous arrays (strided) – look at the comment “masked strided loads. This has a kernel loop and a remainder loop
Avoiding Multiversioned Loops

Fortran: declare the assumed shape arrays are CONTIGUOUS

```fortran
real, contiguous :: theta(:,), sth(:)
```

Avoids the extra test and loop multi-version

What happens if non-contiguous slices are passed?

- Another alternative: pass arrays as assumed-size

```fortran
real theta(*), sth(*)
```

...arguments received are contiguous

- There are many ways to pass array arguments in Fortran. Understand how this affects vectorization strategies:

  • Fortran Array Data and Arguments and Vectorization

Controlling Vectorization of Loops with OpenMP 4 SIMD Directives
OpenMP SIMD Loop Vect Control

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

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

- Now incorporated in OpenMP 4.0 ⇒ portable
  - -qopenmp or -qopenmp-simd to enable

- OMP 4 coming in gfortran 4.9.1 and other vendor compilers
  - Gfortran will need -fopenmp or -fopenmp-simd
SIMD loops: syntax

\texttt{!$omp simd} \texttt{[clauses]}

\textit{DO loop, or array expression}

\texttt{[!$omp end simd]}

Loop has to be in “Canonical loop form”

- Modern F90 DO / END DO (but not DO WHILE)
- Old style DO 100 / 100 CONTINUE
- Array syntax: \( a = b \times c \)  ! Where \( a \), \( b \), \( c \) are arrays
- \texttt{DO CONCURRENT}, OMP loops: will parallelize AND vectorize
- \texttt{WHERE} – may be a candidate. depends on compiler version and target arch
  - Older Intel Fortran compilers – will not vectorize
  - May use array temporaries, compute both versions of results, merge base on mask
- MIC, AVX and AVX-512: use efficient masked vector instructions
Compiler recognition of OMP directives

Compile with either [-q | /Q]openmp or [-q | /Q]openmp-simd

If your code is already using openmp and compiled with [-q | /Q]openmp, the SIMD directives will be recognized. OpenMP will parallel as expected, AND also look for !$omp SIMD directives and vectorize as directed.

If your code ONLY as !$omp SIMD directives (no parallel directives) OR you have both SIMD and parallel directives but only want to interprete the SIMD directives for vectorization compile with [-q | /Q]openmp-simd
Controlling Loop Vectorization

Use !$OMP SIMD in code, compile with -openmp-simd

```fortran
subroutine add(A, N, X)  
  integer N, X  
  real A(N)  
  DO I=X+1, N  
    A(I) = A(I) + A(I-X)  
  ENDDO  
end

Does Not Vectorize
```

```fortran
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

Vectorizes!
```

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)  

Minimizes source code changes needed to enforce vectorization
Clauses for OMP SIMD directives

The programmer (i.e. you!) is responsible for correctness

- Just like for race conditions in loops with OpenMP threading

Available clauses: !$omp simd <clauses>

- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- REDUCTION
- COLLAPSE (for nested loops)
- LINEAR (additional induction variables)
- SAFELEN (max iterations that can be executed concurrently)
- ALIGNED (tells compiler about data alignment)
Data Sharing Clauses

private(var-list):
Uninitialized vectors for variables in var-list – allocate register for X, no initial value in each SIMD lane

firstprivate(var-list):
Initialized vectors for variable(s), each simd lane gets initial value

reduction(op:var-list):
Create private variables for var-list and apply reduction operator op at the end of the construct
SIMD Loop Clauses

aligned (list[:alignment])
- Specifies that the list items have a given alignment
- Default is alignment for the architecture

linear (list[:linear-step])
- The variable’s value is in relationship with the iteration number
  \[ x_i = x_{\text{orig}} + i \times \text{linear-step} \]

safelen (length)
- Maximum number of iterations that can run concurrently without breaking a dependence
- In practice, set to the number of elements in the vector length

collapse (n)
- Collapse perfectly-nested loops into a single iteration space (loop) and vectorize
SIMD loop example

subroutine mypi(count, pi)
    integer, intent(in) :: count
    real , intent(out):: pi
    real :: t
    integer :: i
    pi = 0.0
!
$omp simd private(t) reduction(+:pi)
    do i=1, count
        t = ((i+0.5)/count)
        pi = pi + 4.0/(1.0+t*t)
    end do
    pi = pi/count
end subroutine mypi
SIMD Functions

Assisting the compiler when your loops contain calls to user-written procedures
Why? Procedure calls inside loops break vectorization

- Procedures may introduce loop carry dependencies
- Procedures may have side effects, may need specific serial loop ordering
- Compiler defaults to SAFE non-vector code
- Exception: intrinsic functions, Intel Fortran provides vector-safe versions that are automatically substituted
  - -lib-inline (default) -nolib-inline
Vectorized SIMD intrinsic functions

- Calls to most mathematical functions in a loop body can be vectorized using “Short Vector Math Library”:
  - Short Vector Math Library (libsvml) provides vectorized implementations of different mathematical functions
  - Optimized for latency compared to the VML library component of Intel® MKL which realizes same functionality but which is optimized for throughput

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>ceil</td>
<td>fabs</td>
<td>round</td>
</tr>
<tr>
<td>acosh</td>
<td>cos</td>
<td>floor</td>
<td>sin</td>
</tr>
<tr>
<td>asin</td>
<td>cosh</td>
<td>fmax</td>
<td>sinh</td>
</tr>
<tr>
<td>asinh</td>
<td>erf</td>
<td>fmin</td>
<td>sqrt</td>
</tr>
<tr>
<td>atan</td>
<td>erfc</td>
<td>log</td>
<td>tan</td>
</tr>
<tr>
<td>atan2</td>
<td>erfinv</td>
<td>log10</td>
<td>tanh</td>
</tr>
<tr>
<td>atanh</td>
<td>exp</td>
<td>log2</td>
<td>trunc</td>
</tr>
<tr>
<td>cbrt</td>
<td>exp2</td>
<td>pow</td>
<td></td>
</tr>
</tbody>
</table>

Also float versions, such as sinf()

Random Numbers
C/C++: drand48()
Fortran: random_number()
Concept of SIMD Functions

OMP 4: Allows use of scalar syntax to describe an operation on a single element

Applies operation to arrays in parallel, utilizing vector parallelism (process array elements in blocks of vector length)

The programmer:

- Writes a standard procedure with a **scalar syntax**
- Call the procedure with **scalar arguments** (no change in syntax for the call)
- Annotates procedure at definition with
  
  `!$omp declare SIMD <function, clauses>`

- At procedure call site, alert the compiler to look for a SIMD version of the procedure
  
  `!$omp simd <clauses>`

The compiler:

- Generates a short vector version, and a serial version
- Invokes the vector version in vectorizable regions, serial otherwise
SIMD procedures: Syntax

```c
!$omp declare simd (function-or-procedure-name) [clauses]
```

Instructs the compiler to

- generate a SIMD-enabled version(s) of a given procedure (subroutine or function)
- that a SIMD-enabled version procedure is available to use from a SIMD loop
SIMD functions: clauses

**simdlen**(length)
- generate function to support a given vector length

**uniform**(argument-list)
- argument has a constant value (invariant for all invocations of the procedure in the calling loop)

**inbranch**
- function always called from inside an if statement

**notinbranch**
- function never called from inside an if statement

**linear**(argument-list[:linear-step])
**aligned**(argument-list[:alignment])
**reduction**(operator:list)

Similar to annotations used on loop vars, only applied to arguments
Example

We integrate the function:

\[ f(x) = \frac{4}{1+x^2} \]

between the limits \( x=0 \) and \( x=1 \). The result approximates the value of \( \pi \).

The integration method is the \( n \)-point rectangle quadrature rule.

```fortran
program compute_pi
    integer n, i
    double precision sum, pi, x, h, f
    real start, finish
    external f

    n = 1000000000
    h = 1.0/n
    sum = 0.0

    call cpu_time(start)
    do 10 i = 1,n
        x = h*(i-0.5)
        sum = sum + f(x)
    10 continue
    pi = h*sum
end
```

double precision function f(x)

double precision x

\[
f = \frac{4}{1+x^2}
\]

end

ifort -O2 -xhost -c fx.f
ifort -O2 -xhost -vec-report3 pi.f fx.o

Example: SIMD function

```fortran
program compute_pi
  integer n, i
  double precision sum, pi, x, h
  real start, finish

  interface
    double precision function f(x)
  end interface

  !$omp declare simd f
  double precision x
  end
  end interface

  n = 1000000000
  h = 1.0/n
  sum = 0.0

  call cpu_time(start)
  !$omp simd reduction(+:sum)
  do 10 i = 1,n
    x = h*(i-0.5)
    sum = sum + f(x)
  10 continue
  pi = h*sum

!

!)omp declare simd f
  double precision function f(x)
  double precision x
    f = (4/(1+x*x))
  end

ifort -O2 -xhost -c fx2.f
ifort -O2 -xhost -vec-report3 pi2.f fx2.o

pi2.f(43): (col. 12) remark: SIMD LOOP WAS VECTORIZED
```
Example: module function

program computepi
    use func_f
    integer       n, i
    double precision  sum, pi, x, h
    real start, finish

    n = 1000000000
    h = 1.0/n
    sum = 0.0

    call cpu_time(start)
    !$omp simd reduction(+:sum)
    do 10 i = 1,n
       x = h*(i-0.5)
       sum = sum + f(x)
    10     continue
    pi = h*sum
end program computepi

module func_f
contains
    !$omp declare simd f
    function f(x)
        real(8) :: f
        real(8) :: x
        f = (4/(1+x*x))
    end function
end module func_f

ifort -O2 -xhost -c fx3.f
ifort -O2 -xhost -vec-report3 pi3.f fx3.o

pi2.f(43): (col. 12) remark: SIMD LOOP WAS VECTORIZED

module USE brings in the SIMD attribute and definition of function ‘f’
Restrictions on Fortran SIMD procedures

*proc-name* must not be a generic name, procedure pointer or entry name.

Any *declare simd* directive must appear in a specification part of a subroutine subprogram, function subprogram or interface body to which it applies.

If a *declare simd* directive is specified in an interface block for a procedure, it must match a *declare simd* directive in the definition of the procedure.

If a procedure is declared via a procedure declaration statement, the procedure *proc-name* should appear in the same specification.
OpenMP* 4.0 SIMD vs. Intel® SIMD

<table>
<thead>
<tr>
<th>OpenMP* 4.0 SIMD</th>
<th>Intel® SIMD</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp simd [clauses] for-loops</td>
<td>#pragma simd [clauses] for-loops</td>
<td>Enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions.</td>
</tr>
<tr>
<td>!$omp simd [clauses] do-loops</td>
<td>!$dec$ simd [clauses] do-loops</td>
<td></td>
</tr>
<tr>
<td>[$!omp end simd]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clauses: private, lastprivate, reduction, linear, safelen, collapse</td>
<td>Clauses: private, firstprivate, lastprivate, reduction, linear, vectorlength</td>
<td></td>
</tr>
<tr>
<td>#pragma omp for simd [clauses]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma omp parallel for simd [clauses]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!$omp do simd [clauses]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!$omp parallel do simd [clauses]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma omp declare simd [clauses]</td>
<td>__declspec(vector([clauses])) (Windows*)</td>
<td>Multiple invocations may be executed concurrently</td>
</tr>
<tr>
<td>function-declaration-or-definition</td>
<td><strong>attribute</strong>((vector([clauses]))) (Linux*)</td>
<td></td>
</tr>
<tr>
<td>!$omp declare simd (proc-name) [clauses]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>simdlen(length)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear(argument-list[:linear-step])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aligned(argument-list[:alignment])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uniform(argument-list)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reduction(operator:list)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inbranch</td>
<td>notinbranch</td>
<td></td>
</tr>
<tr>
<td>_dir$ attributes vector [clause[[], clause] ...] :: proc-name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vectorlength(length)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear(argument-list[:linear-step])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aligned(argument-list[:alignment])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uniform(argument-list)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mask</td>
<td>nomask</td>
<td></td>
</tr>
</tbody>
</table>
Vectorization Summary

The importance of SIMD parallelism is increasing

- Moore’s law leads to wider vectors as well as more cores
- Don’t leave performance “on the table”
- Be ready to help the compiler vectorizer if necessary
  - With OpenMP 4 SIMD Loop directives
  - With OpenMP 4 SIMD procedures

No need to re-optimize vectorizable code for new processors

- Typically a simple recompilation
Further Information on vectorization

The Intel® Compiler User Guides:
https://software.intel.com/en-us/compiler_15.0_ug_f

Series of short, audio-visual vectorization tutorials:

New Optimization Report (compilers version 15.0+)

Other articles:
• Requirements for Vectorizable Loops
http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops

• Explicit Vector Programming in Fortran

• Fortran Array Data and Arguments and Vectorization
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Vectorization – More Switches and Directives

Disable vectorization

Globally via switch: \{L&M\}: \texttt{-no-vec\{W\}: /Qvec-}

For a single loop: directive \texttt{!DIR$ novector}

- Disabling vectorization means not using \texttt{packed SSE/AVX} instructions.
- The compiler still may use SSE instruction set extensions

Enforcing vectorization for a loop - overwriting the compiler heuristics: \texttt{!dir$ vector always}

- will enforce vectorization even if the compiler thinks it is not profitable to do so (e.g due to non-unit strides or alignment issues)

- \textbf{It’s a SUGGESTION: Will not enforce vectorization} if the compiler fails to recognize this as a semantically correct transformation – hint to compiler to relax

- Using directive \texttt{!dir$ vector always assert} will print error message in case the loop cannot be vectorized and will abort compilation
Vectorization – Examples in Fortran
Will it vectorize? (Fortran)

Assume a, b and x are known to be independent.

```fortran
do j=1,n
    a(j) = a(j-k) + b(j)
  enddo

do i=1,n,4
    a(i) = b(i) + x(i)
  enddo

do j=1,n
  do i=1,n
    b(i) = b(i) + x(j) * a(j,i)
  enddo
enddo

do i=1,n
  b(i) = b(i) + a(i)*x(index(i))
enddo

do j=1,n
  sum = sum + a(j)*b(j)
enddo

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
```
Will it vectorize? Answers

1) Vectorizes if \( k \leq 0 \); doesn’t vectorize if \( k > 0 \) and small; may vectorize if \( k \geq \) number of elements in a vector register

2) Unlikely to vectorize because of non-unit stride (inefficient)

3) Doesn’t vectorize because of non-unit stride, unless compiler can first interchange the order of the loops. (Here, it can)

4) Usually doesn’t vectorize because of indirect addressing (non-unit stride), would be inefficient. If \( x(index(i)) \) appeared on the LHS, this would also introduce potential dependency (\( index(i) \) might have the same value for different values of \( i \))

5) Reductions such as this will vectorize. The compiler accumulates a number of partial sums (equal to the number of elements in a vector register), and adds them together at the end of the loop.

6) This will vectorize. Neither “if” masks nor most simple math intrinsic functions prevent vectorization. But with SSE, the sqrt is evaluated speculatively. If FP exceptions are unmasked, this may trap if \( s<0 \), despite the if clause. With AVX, there is a real hardware mask, so the sqrt will never be evaluated if \( s<0 \), and no exception will be trapped.