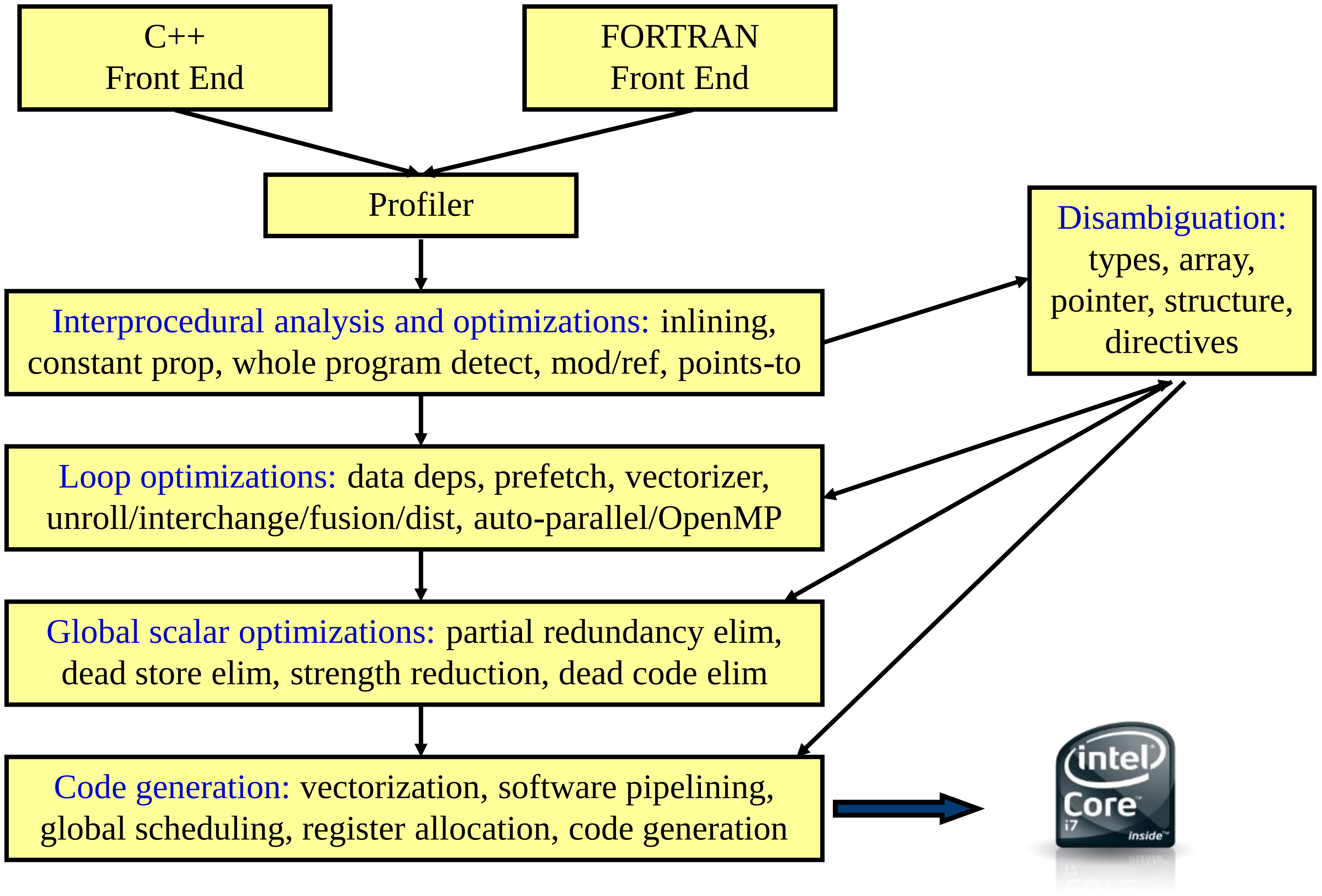
INTEL'S OPTIMIZING COMPILER

Kenneth Craft
Technical Consulting Engineer
Intel® Corporation

Intel® Compiler Architecture


C++ Front End
FORTRAN Front End
Profiler
Interprocedural analysis and optimizations: inlining, constant prop, whole program detect, mod/ref, points-to
Disambiguation: types, array, pointer, structure, directives
Loop optimizations: data deps, prefetch, vectorizer, unroll/interchange/fusion/dist, auto-parallel/OpenMP
Global scalar optimizations: partial redundancy elim, dead store elim, strength reduction, dead code elim
Code generation: vectorization, software pipelining, global scheduling, register allocation, code generation
intel Core i7 inside

Getting Started

Set the environment, e.g.:

- source compilervars.sh intel64   (or .csh)   Linux* or OS* X
- compilervars.bat intel64   (or ia32)   Windows*

Drivers:

- ifort   for Fortran on all OS
  - assumes .f90 free format, .f fixed format by default
- icl   on Windows
- icc   for C, icpc for C++ on Linux

  but treats .c as C and .cpp as C++ by default

Linking:

- Simplest: use compiler drivers above
  - links Intel-specific libraries, such as optimized math functions
- else use xild (xilink), which invokes ld (link) on Linux (Windows)

Or use the Microsoft* Visual Studio* IDE on Windows.

General Compiler Features

Supports standards

- Fortran77, Fortan90, Fortran95, Fortran2003, much Fortran 2008
- Up to C99, C++11; Some C++ 14; Minimal C11 (so far)
  - -std=c99 -std=c++11 -std=c++14

Intel® Fortran (and C/C++) binary compatible with gcc, gdb, …

- But <u>not</u> binary compatible with gfortran

Supports all instruction sets via vectorization (auto- and explicit)

- Intel® SSE, Intel® AVX, Intel® AVX-512, Intel® MIC Architecture

OpenMP* 4.0 support, some 4.5, no Fortran user-defined reductions

Optimized math libraries

Many advanced optimizations

- With detailed, structured optimization reports

Some Useful Features of **ifort**
(that you might not know)

-assume buffered_io (/assume:buffered_io)

- Can improve I/O performance by buffering
- Can also enable with export FORT_BUFFERED=true or via OPEN statement

-convert big_endian (/convert:big_endian)

- Converts data from/to big endian format on input/output

-cxxlib

- link to default C++ RTL (for mixed language apps); Linux* only

-fpp (/fpp)

- Preprocess before compilation
- default for .F, .F90 files (Linux), but not for .f, .f90, assumed preprocessed

-mkl (/mkl)

- Link the Math Kernel Library

-traceback (/traceback)

- For low-overhead stack trace in case of runtime failure

Basic Optimizations with ifort -O…

-O0 no optimization; sets -g for debugging

-O1 scalar optimizations

- Excludes optimizations tending to increase code size

-O2 **default** for ifort (except with -g)

- includes **auto-vectorization**; some loop transformations such as unrolling; inlining within source file;
- Start with this (after initial debugging at -O0)

-O3 more aggressive loop optimizations

- Including cache blocking, loop fusion, loop interchange, …
- May not help all applications; need to test

gfortran, gcc **default** is less or no optimization

-O3 includes **vectorization** and most inlining

Intel® Compilers: Loop Optimizations

ifort (or icc or icpc or icl) **-O3**

Loop optimizations:

▪ **Automatic vectorization**‡	(use of packed SIMD instructions)
▪ Loop interchange ‡	(for more efficient memory access)
▪ Loop unrolling‡	(more instruction level parallelism)
▪ Prefetching	(for patterns not recognized by h/w prefetcher)
▪ Cache blocking	(for more reuse of data in cache)
▪ Loop versioning ‡	(for loop count; data alignment; runtime dependency tests)
▪ Memcpy recognition ‡	(call Intel's fast memcpy, memset)
▪ Loop splitting ‡	(facilitate vectorization)
▪ Loop fusion	(more efficient vectorization)
▪ Scalar replacement‡	(reduce array accesses by scalar temps)
▪ Loop rerolling	(enable vectorization)
▪ Loop peeling ‡	(allow for misalignment)
▪ Loop reversal	(handle dependencies)
▪ etc.	

‡ all or partly enabled at -O2

Processor-specific Compiler Switches

Intel® processors only	Intel and non-Intel (-m also GCC)
-xsse2	-msse2 (default)
-xsse3	-msse3
-xssse3	-mssse3
-xsse4.1	-msse4.1
-xsse4.2	-msse4.2
-xavx	-mavx
-xcore-avx2	
-xmic-avx512	
-xHost	-xHost (-march=native)
Intel cpuid check	No cpu id check
Runtime message if run on unsupported processor	Illegal instruction error if run on unsupported processor

Processor Dispatch (fat binaries)

Compiler can generate multiple code paths

- optimized for different processors
- only when likely to help performance
- One default code path, one or more optimized paths
- Optimized paths are for Intel processors only
- Default code path can be modified using switches from preceding slide

Examples:

- -axavx
  - default path optimized for Intel® SSE2 (Intel or non-Intel)
  - Second path optimized for Intel® AVX (code name Sandy Bridge, etc.)
- -axcore-avx2,avx -xsse4.2
  - Default path optimized for Intel® SSE4.2 (code name Nehalem, Westmere)
  - Second path optimized for Intel® AVX (code name Sandy Bridge, etc.)
  - Third path optimized for Intel® AVX2 (code name Haswell)

InterProcedural Optimization (IPO)

ifort -ipo

Analysis & Optimization across function and source file boundaries, e.g.

- Function inlining; Interprocedural constant propagation; Alignment analysis; Disambiguation; Data & Function Layout; etc.

2-step process:

- Compile phase – objects contain intermediate representation
- “Link” phase – compile and optimize over all such objects
  - Fairly seamless: the linker automatically detects objects built with -ipo, and their compile options
  - May increase build-time and binary size
  - But can be done in parallel with -ipo=n
  - Entire program need not be built with IPO/LTO, just hot modules

Particularly effective for apps with many smaller functions

Get report on inlined functions with -qopt-report-phase=ipo

Math Libraries

icc (ifort) comes with optimized math libraries

- libimf (scalar) and libsvml (vector)
- Faster than GNU libm
- Driver links libimf automatically, ahead of libm
- More functionality (replace math.h by mathimf.h for C)

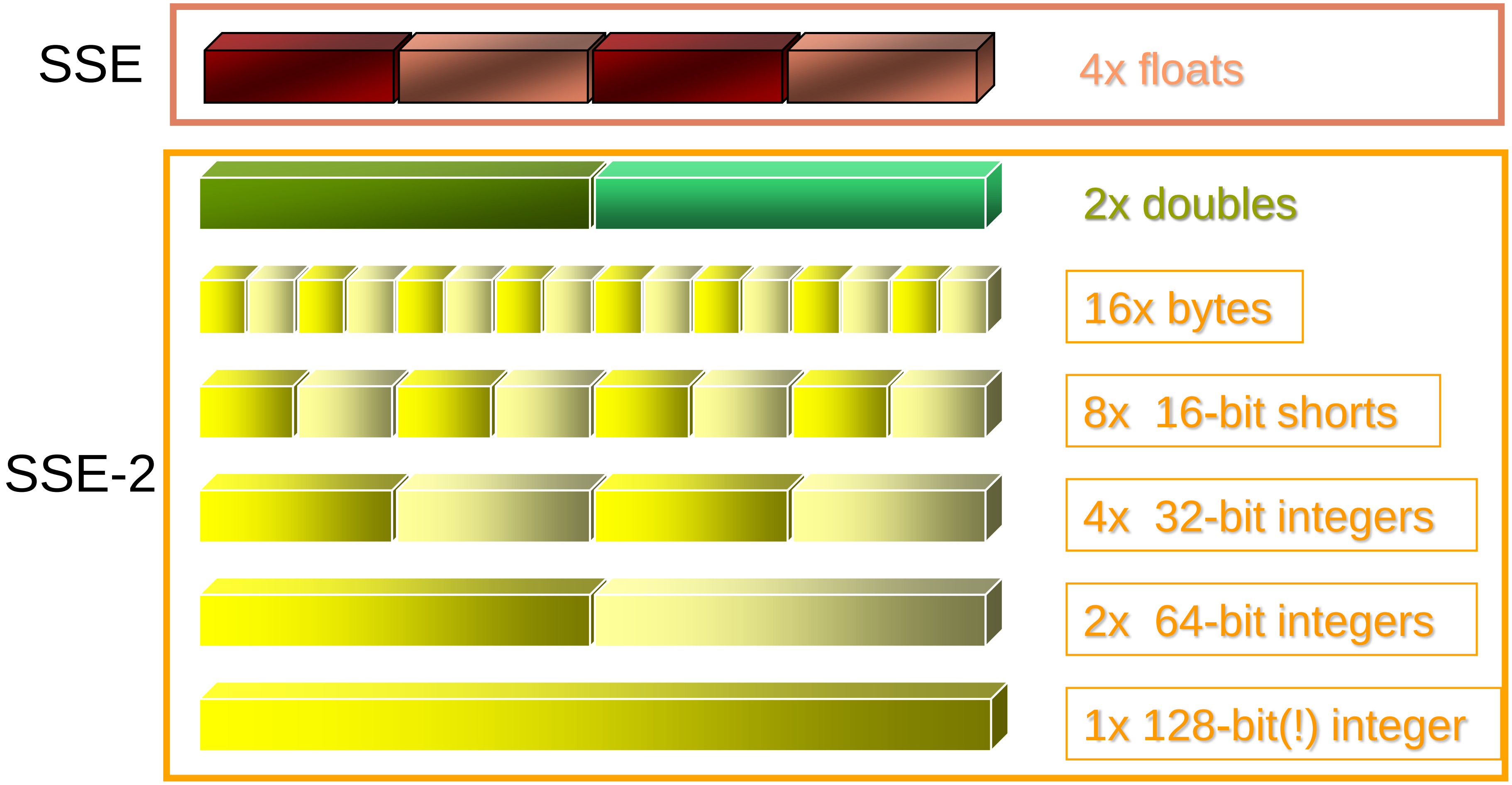
Don't link to libm explicitly! 🚫 -lm 🚫

- May give you the slower libm functions instead
- Though the Intel driver may try to prevent this
- GCC needs -lm, so it is often found in old makefiles

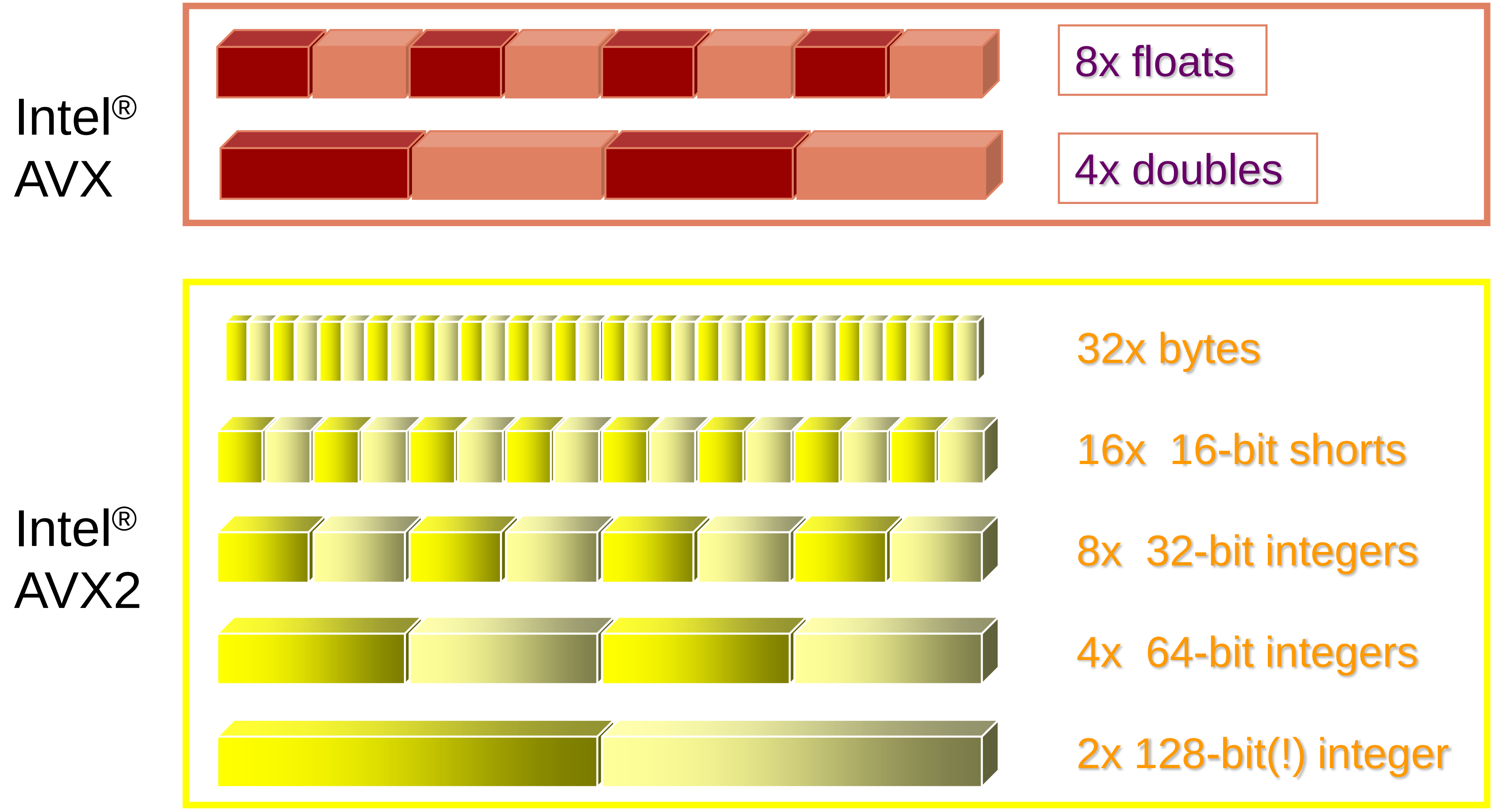
Low precision option for vectorized math library may be faster, if precision is sufficient

- For doubles, still more accurate than single precision
- -fimf-precision=low

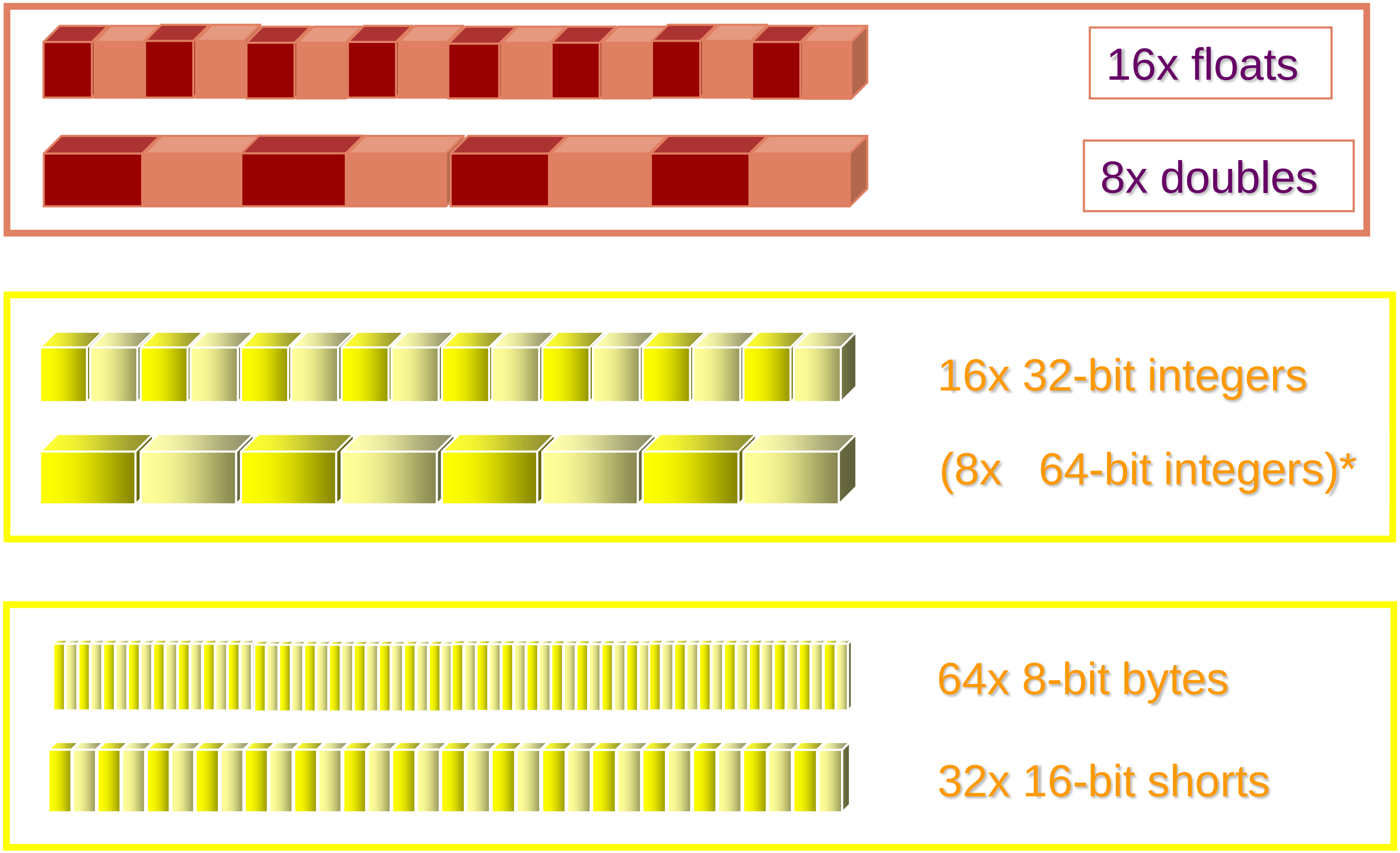
Intel® SSE & Intel® AVX-128 Data Types



Intel® Advanced Vector Extensions 2 (Intel® AVX2) Data Types



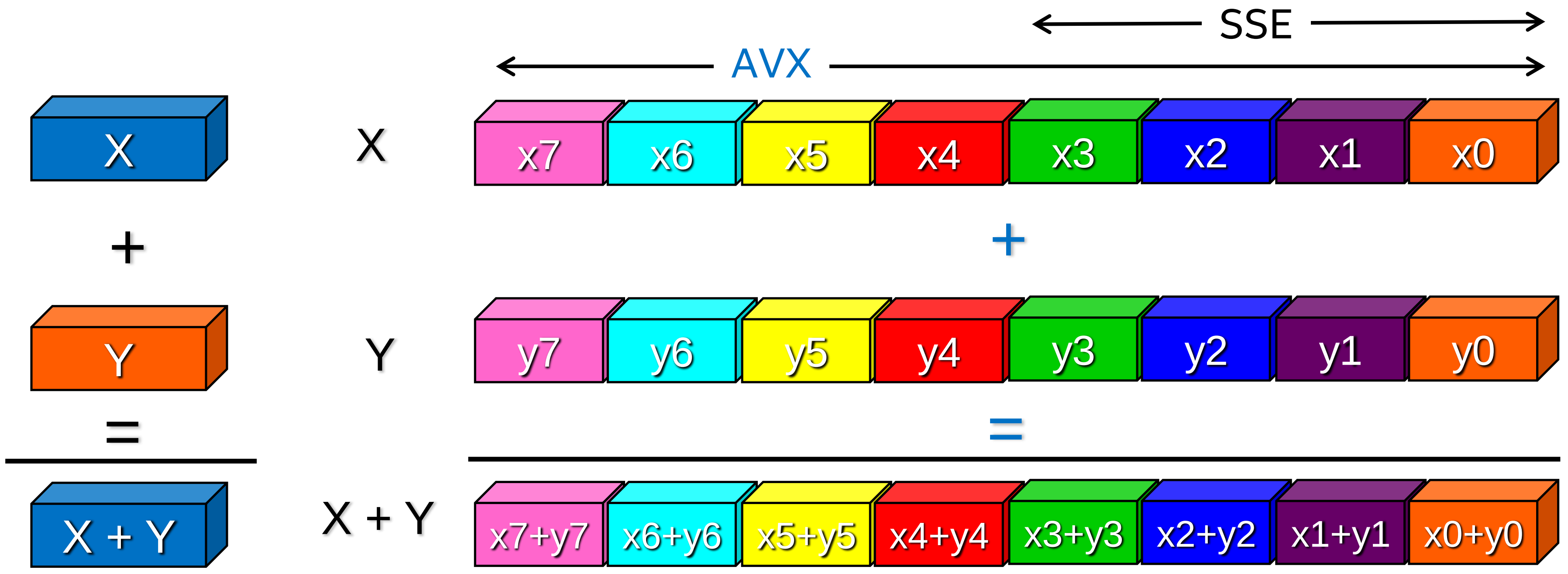
Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Data Types



SIMD: Single Instruction, Multiple Data

```
for (i=0;  i<n;  i++)      z[i] = x[i] + y[i];
```

- Scalar mode
  - one instruction produces one result
  - E.g. vadd**ss**, (vadd**sd**)
- Vector (SIMD) mode
  - one instruction can produce multiple results
  - E.g. vadd**ps**, (vadd**pd**)



Vectorizable math functions

acos	ceil	fabs	pow
acosh	cos	floor	round
asin	cosh	fmax	sin
asinh	erf	fmin	sinh
atan	erfc	fmod	sqrt
atan2	erfinv	log	tan
atanh	exp	log10	tanh
cbrt	exp2	log2	trunc

Also float versions and Fortran equivalents

Uses short vector math library, libsvml

Entry points such as
Intel® SSE:
__svml_sin2
__svml_sinf4
Intel® AVX:
__svml_pow4
__svml_powf8

Many routines in the libsvml math library are more highly optimized for Intel microprocessors than for non-Intel microprocessors.

Compiling for Intel® AVX (high level)

Compile with -xavx (Intel® AVX; Sandy Bridge, Ivy Bridge, etc.)

Compile with -xcore-avx2 (Intel® AVX2; Haswell, Broadwell)

- Intel processors only (-mavx, -march=core-avx2 for non-Intel)
- Vectorization works just as for Intel® SSE, but with longer vectors
  - More efficient loads & stores if data are 32 byte aligned
- More loops can be vectorized than with SSE
  - Individually masked data elements
  - More powerful data rearrangement instructions

-axavx (-axcore-avx2) gives both SSE2 and AVX (AVX2) code paths

- use -x or -m switches to modify the default SSE2 code path
  - Eg -axcore-avx2 -xavx to target both Haswell and Sandy Bridge

Math libraries may target AVX and/or AVX2 automatically at runtime

Some Benefits of Intel® AVX

For Intel® AVX, main speedups are for floating point

- Wider SIMD floating-point instructions ⇒ better throughput
    - Except double precision division or square root
- Enhanced data rearrangement instructions
- Non-destructive, 3 operand syntax

For Intel® AVX2, main additional speedups come from

- Wider SIMD integer instructions
- Fused multiply-add instructions
- Gather & permute instructions enable more vectorization
- More efficient general 32 byte loads (less split loads)

Applications most likely to benefit

- Spend much time in vectorizable loops
- Are not memory bound

Factors that have Impact on Vectorization

Loop-carried dependencies

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

Unknown/aliased loop iteration count

```
struct _x { int d; int bound; };

void doit(int *a, struct _x *x)
{
  for(int i = 0; i < x->bound; i++)
    a[i] = 0;
}
```

Function calls

```
for (i = 1; i < nx; i++) {
  x = x0 + i * h;
  sumx = sumx + func(x, y, xp);
}
```

Indirect memory access

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

Pointer aliasing

```
void scale(int *a, int *b)
{
   for (int i = 0; i < 1000; i++)
          b[i] = z * a[i];
}
```

Outer loops

```
for(i = 0; i <= MAX; i++) {
  for(j = 0; j <= MAX; j++) {
    D[i][j] += 1;
  }
}
```

many ……

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"

- Without help, 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
  - to 32 byte boundaries (for Intel® AVX instructions)
  - else to 16 bytes, or at least to "natural" alignment

How to Align Data (Fortran)

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

```
!dir$ attributes align:n :: array
```

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

For a 2D array, choose column length to be a multiple of n,
so that consecutive columns have the same alignment (pad if necessary)

`-align array32byte` compiler tries to align all array types

And tell the compiler…

```
!dir$ vector aligned   OR
!$omp simd aligned( var [,var…]:<n>)
```

- Asks compiler to vectorize, 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, …]
```

- Compiler may assume array is aligned to n byte boundary
  - Typical use is for dummy arguments
  - Extension for allocatable arrays in next compiler version

n=16 for Intel® SSE, n=32 for Intel® AVX, n=64 for Intel® AVX-512

How to Align Data (C/C++)

Allocate memory on heap aligned to n byte boundary:

```
void* _mm_malloc(int size, int n)
int posix_memalign(void **p, size_t n, size_t size)
void* aligned_alloc(size_t alignment, size_t size)    (C11)
#include <aligned_new>                                (C++11)
```

Alignment for variable declarations:

```
__attribute__((aligned(n)))  var_name      or
__declspec(align(n))  var_name
```

And tell the compiler…

```
#pragma 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

```
__assume_aligned(array, n)
```

- Compiler may assume array is aligned to n byte boundary

n=64 for Intel® Xeon Phi™ coprocessors, n=32 for Intel® AVX, n=16 for Intel® SSE

Problems with Pointers

Hard for compiler to know whether arrays or pointers might be aliased (point to the same memory location)

- Aliases may hide dependencies that make vectorization unsafe
- Bigger problem for C than Fortran (which has TARGET attribute)
  - ASSOCIATE (Fortran 2003) can result in aliasing

In simple cases, compiler may generate vectorized and unvectorized loop versions, and test for aliasing at runtime

Otherwise, compiler may need help:

- -fargument-noalias & similar switches; “restrict” keyword for C
- !dir$ ivdep   asserts no potential dependencies
  - Compiler still checks for proven dependencies
- !$OMP SIMD   asserts no dependencies, period   (see later)
- Prefer allocatable arrays to pointers where possible !

```
Real, pointer, dimension(:) :: v, w, x, y, z
!dir$ivdep
 do i=1, n
     z(i) = v(i)*w(i) + x(i)*y(i)
```

Intel® Compilers:
some useful loop optimization pragmas/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
▪ BLOCK_LOOP	suggest blocking factor for more efficient use of cache
▪ UNROLL_AND_JAM	increase amount of work per iteration of inner loop

Use where needed to help the compiler,
guided by optimization reports

Hierarchical Loop Optimization Report

Peel loop, remainder loop and kernel

```
LOOP BEGIN at ggFineSpectrum.cc(124,5) inlined into ggFineSpectrum.cc(56,7)
   remark #15018: loop was not vectorized: not inner loop

   LOOP BEGIN at ggFineSpectrum.cc(138,5) inlined into ggFineSpectrum.cc(60,15)
   Peeled
      remark #25460: Loop was not optimized
   LOOP END

   LOOP BEGIN at ggFineSpectrum.cc(138,5) inlined into ggFineSpectrum.cc(60,15)
      remark #15145: vectorization support: unroll factor set to 4
      remark #15002: LOOP WAS VECTORIZED
   LOOP END

   LOOP BEGIN at ggFineSpectrum.cc(138,5) inlined into ggFineSpectrum.cc(60,15)
   Remainder
       remark #15003: REMAINDER LOOP WAS VECTORIZED
   LOOP END
LOOP END
```

Outer loop of nest

"Peel" loop for data alignment

Vectorized loop kernel

Remainder loop

Example of New Optimization Report

```
$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr foo.c

Begin optimization report for: foo

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(4,3)
Multiversioned v1
   remark #25231: Loop multiversioned for Data Dependence
   remark #15135: vectorization support: reference theta has unaligned access
   remark #15135: vectorization support: reference sth has unaligned access
   remark #15127: vectorization support: unaligned access used inside loop body
   remark #15145: vectorization support: unroll factor set to 2
   remark #15164: vectorization support: number of FP up converts: single to double precision 1
   remark #15165: vectorization support: number of FP down converts: double to single precision 1
   remark #15002: LOOP WAS VECTORIZED
   remark #36066: unmasked unaligned unit stride loads: 1
   remark #36067: unmasked unaligned unit stride stores: 1
   ….  (loop cost summary)  ….
   remark #25018: Estimate of max trip count of loop=32
LOOP END

LOOP BEGIN at foo.c(4,3)
Multiversioned v2
   remark #15006: loop was not vectorized: non-vectorizable loop instance from multiversioning
LOOP END
=================================================================
```

```c
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 128; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

Optimization Report Example

```
$ icc -c  -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias foo.c
Begin optimization report for: foo
   Report from: Loop nest & Vector optimizations [loop, vec]
```

(/Qalias-args- on Windows*)

```
LOOP BEGIN at foo.c(4,3)
  remark #15135: vectorization support: reference theta has unaligned access
  remark #15135: vectorization support: reference sth has unaligned access
  remark #15127: vectorization support: unaligned access used inside loop body
  remark #15145: vectorization support: unroll factor set to 2
  remark #15164: vectorization support: number of FP up converts: single to double precision 1
  remark #15165: vectorization support: number of FP down converts: double to single precision 1
  remark #15002: LOOP WAS VECTORIZED
  remark #36066: unmasked unaligned unit stride loads: 1
  remark #36067: unmasked unaligned unit stride stores: 1
  remark #36091: --- begin vector loop cost summary ---
  remark #36092: scalar loop cost: 114
  remark #36093: vector loop cost: 55.750
  remark #36094: estimated potential speedup: 2.040
  remark #36095: lightweight vector operations: 10
  remark #36096: medium-overhead vector operations: 1
  remark #36098: vectorized math library calls: 1
  remark #36103: type converts: 2
  remark #36104: --- end vector loop cost summary ---
  remark #25018: Estimate of max trip count of loop=32
LOOP END
```

```
#include <math.h>
void foo (float * theta, float * sth)   {
  int i;
  for (i = 0; i < 128; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

Optimization Report Example

```
$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias foo.c

Begin optimization report for: foo

   Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(4,3)
remark #15135: vectorization support: reference theta has unaligned access
  remark #15135: vectorization support: reference sth has unaligned access
  remark #15127: vectorization support: unaligned access used inside loop body
  remark #15002: LOOP WAS VECTORIZED
  remark #36066: unmasked unaligned unit stride loads: 1
  remark #36067: unmasked unaligned unit stride stores: 1
  remark #36091: --- begin vector loop cost summary ---
  remark #36092: scalar loop cost: 111
  remark #36093: vector loop cost: 28.000
  remark #36094: estimated potential speedup: 3.950
  remark #36095: lightweight vector operations: 9
  remark #36098: vectorized math library calls: 1
  remark #36104: --- end vector loop cost summary ---
  remark #25018: Estimate of max trip count of loop=32
LOOP END
```

```c
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 128; i++)
     sth[i] = sinf(theta[i]+3.1415927f);
}
```

Optimization Report Example

```
$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias -xavx
foo.c

Begin report for: foo

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(4,3)
   remark #15135: vectorization support: reference theta has unaligned access
   remark #15135: vectorization support: reference sth has unaligned access
   remark #15127: vectorization support: unaligned access used inside loop body
   remark #15002: LOOP WAS VECTORIZED
   remark #36066: unmasked unaligned unit stride loads: 1
   remark #36067: unmasked unaligned unit stride stores: 1
   remark #36091: --- begin vector loop cost summary ---
   remark #36092: scalar loop cost: 110
   remark #36093: vector loop cost: 15.370
   remark #36094: estimated potential speedup: 7.120
   remark #36095: lightweight vector operations: 9
   remark #36098: vectorized math library calls: 1
   remark #36104: --- end vector loop cost summary ---
   remark #25018: Estimate of max trip count of loop=16
LOOP END
===========================================================================
```

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  for (i = 0; i < 128; i++)
     sth[i] = sinf(theta[i]+3.1415927f);
}
```

Optimization Report Example

```
$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias -xavx
foo.c

Begin optimization report for: foo

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(6,3)
remark #15134: vectorization support: reference theta has aligned access
   remark #15134: vectorization support: reference sth has aligned access
   remark #15002: LOOP WAS VECTORIZED
   remark #36064: unmasked aligned unit stride loads: 1
   remark #36065: unmasked aligned unit stride stores: 1
   remark #36091: --- begin vector loop cost summary ---
   remark #36092: scalar loop cost: 110
   remark #36093: vector loop cost: 13.620
   remark #36094: estimated potential speedup: 8.060
   remark #36095: lightweight vector operations: 9
   remark #36098: vectorized math library calls: 1
   remark #36104: --- end vector loop cost summary ---
   remark #25018: Estimate of max trip count of loop=16
LOOP END
===========================================================================
```

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  __assume_aligned(theta,32);
  __assume_aligned(sth,32);
  for (i = 0; i < 128; i++)
    sth[i] = sinf(theta[i]+3.1415927f);
}
```

Optimization Report Example

```
$ icc -c -qopt-report=4 -qopt-report-phase=loop,vec -qopt-report-file=stderr -fargument-noalias -xavx
foo.c

Begin optimization report for: foo

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(7,3)
remark #15134: vectorization support: reference theta has aligned access
   remark #15134: vectorization support: reference sth has aligned access
   remark #15002: LOOP WAS VECTORIZED
   remark #36064: unmasked aligned unit stride loads: 1
   remark #36065: unmasked aligned unit stride stores: 1
   remark #36083: unmasked aligned streaming stores: 1
   remark #36091: --- begin vector loop cost summary ---
   remark #36092: scalar loop cost: 110
   remark #36093: vector loop cost: 13.620
   remark #36094: estimated potential speedup: 8.070
   remark #36095: lightweight vector operations: 9
   remark #36098: vectorized math library calls: 1
   remark #36104: --- end vector loop cost summary ---
   remark #25018: Estimate of max trip count of loop=250000
   remark #15158: vectorization support: streaming store was generated for sth
LOOP END
===========================================================================
```

```
#include <math.h>
void foo (float * theta, float * sth)  {
  int i;
  __assume_aligned(theta,32);
  __assume_aligned(sth,32);
for (i = 0; i < 2000000; i++)
    sth[i] = sinf(theta[i]+3.1415927f);
}
```

Multi-dimensional Arrays

This small function multiplies a matrix **a** by a vector **b**

```
subroutine matvec(m,n,a,b,c)
  integer :: m, n, i, j
  real     :: a(m,n), b(m), c(n)
    do j=1,n
       c(j) = 0.
       do i=1,m
          c(j) = c(j) + a(i,j) * b(i)
       enddo
    enddo
end subroutine matvec
```

```
ifort -c -r8 -O2 -xavx mv.F90 -qopt-report-phase=vec -qopt-report=4
…
 LOOP BEGIN at mv.F90(11,6)
   remark #15300: LOOP WAS VECTORIZED
```

- For C, we would need to assert that a did not overlap b or c
  - For Fortran, the compiler can assume this unless these are pointer arrays

Multi-dimensional Arrays

```
ifort -c -r8 -xavx mv.F90 -qopt-report-phase=vec -qopt-report=4
...
   LOOP BEGIN at mv.F90(11,6)
   <Peeled loop for vectorization>
   LOOP END

   LOOP BEGIN at mv.F90(11,6)
      remark #15389: vectorization support: reference a has unaligned access   [ mv.F90(12,9) ]
      remark #15388: vectorization support: reference b has aligned access   [ mv.F90(12,9) ]
      remark #15305: vectorization support: vector length 4
      remark #15300: LOOP WAS VECTORIZED
...
```

To align **b**, because compiler doesn't know its alignment

Compiler can adjust ("peel") at run-time to align accesses to one array (**b**)
Accesses to **a** remain unaligned.

If you know that **b** and **a** are aligned, you can help the compiler:

To align arrays to 32 bytes (for Intel® AVX):

```
!dir$ attributes align:32 :: a, b
```

Or compile with -align array32byte
For Intel® MIC Architecture, align to 64 bytes instead of 32

Multi-dimensional Arrays - alignment

```
!dir$ assume_aligned a:32, b:32
```

This tells the compiler that the starts of the arrays are aligned

If we add these to the source code,

- the "peel" loop disappears
- **b** has aligned access
- **a** still has unaligned access. Why?

The start of the **first** column of a is aligned. But what about the other columns?

- Only if column length is a multiple of 32 bytes (4 doubles)
- We can assert this to the compiler:

```
!dir$ assume (mod(m,4).eq.0)
```

Pad the first dimension if necessary
E.g. a(31,31) becomes a(32,31)
See driver.F90

```
LOOP BEGIN at mv.F90(11,6)
   remark #15388: vectorization support: reference a has aligned access   [ mv.F90(12,9) ]
   remark #15388: vectorization support: reference b has aligned access   [ mv.F90(12,9) ]
   remark #15305: vectorization support: vector length 4
   remark #15300: LOOP WAS VECTORIZED
```

Multi-dimensional arrays - performance

Optimization Options	Speed-up (without padding)	Speed-up (with padding)
-O1 -xavx	1.0	0.98
-O2 -xavx	2.0	3.2
-O2 -xavx -align array32byte Assume start of **b** and **a** aligned	2.1	3.4
-O2 -xavx -align array32byte Assume start of **b** and **a** aligned Assume column length is multiple of 32 bytes (consecutive columns aligned)	May fail on some platforms, since assumption untrue	4.2

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 2nd Generation Intel® Core™ i7-2600K system, frequency 3.4 GHz, running Ubuntu Linux* version 10.04.4 and using the Intel® Fortran Compiler version 16.0 update 1.

Multi-dimensional Arrays

Final version of example code:

```
subroutine matvec(m,n,a,b,c)
  integer :: m, n, i, j
  real     :: a(m,n), b(m), c(n)
!dir$ assume_aligned a:32, b:32
!dir$ assume (mod(m,4).eq.0)
  do j=1,n
     c(j) = 0.
     do i=1,m
        c(j) = c(j) + a(i,j) * b(i)
     enddo
  enddo
end subroutine matvec
```

ifort -r8 -O2 -xavx mv.F90 -qopt-report=4 -qopt-report-phase=vec

(add -qopt-report-file=stderr to get report on stderr instead of mv.optrpt)

Search Loops

Do calculation until first negative value of **a** is encountered:

```
integer function search(a,b,c,n)
  real, dimension(n) :: a, b, c
  integer            :: n, i

  do i=1,n
     if(a(i).lt.0.) exit
     c(i) = sqrt(a(i)) * b(i)
  enddo

  search = i-1
end function search
```

```
ifort -c -qopt-report-file=stderr -qopt-report-phase=vec search3.f90
...
  remark #15520: loop was not vectorized: loop with multiple exits cannot be vectorized unless it meets search loop idiom criteria
```

- Compiler can recognize and vectorize only pristine search loops
  - If “search” idiom not recognized, SIMD directives won’t help

Search Loops - autovectorized

Need to split into pure search loop and calculational loop:

```
integer function search(a,b,c,n)
  real, dimension(n) :: a, b, c
  integer            :: n, i, j

  do i=1,n
     if(a(i).lt.0.) exit
  enddo
 search = i-1

  do j=1,search
     c(j) = sqrt(a(j)) * b(j)
  enddo

end function search
```

- Search loop is recognized; both loops get vectorized
  - Good speed up provided number of executed iterations is not too small.

Fortran Assumed Shape Array Arguments may not be contiguous

```
subroutine func( a, b, n )
  real       :: a(:), b(:)
  integer   :: i, n

  do i=1,n
     b(i) = b(i) + a(i) * 2.
  end do
end
```

<**Multiversioned v1**>
  remark #25233: **Loop multiversioned for stride tests on Assumed shape arrays**

One version has unit stride loads, one has gathers.
In more complex cases, may prevent vectorization

If arguments are contiguous, tell the compiler:

Real,   **contiguous**   :: **a(:), b(:)**   (or real :: a(*), b(*))

Indirect addressing

Not so long ago, loops such as these would not have been vectorized:

```
for (i = 0; i < m; i++ ) {
        b[i] = a[ind[i]] + … ;
}
```

```
do i=1, m
   b(i) = a(ind(i)) + …
enddo
```

The compiler can issue vector stores for **b**, but not vector loads for **a**

Recent compilers can vectorize the rest of the loop, even when issuing separate loads for **a(ind(i))**

The following example calculates a net short-range potential by looping over a list of other particles that are within the range of the potential

```
do jnear=1,ni
       jpt  = ptind(jnear,ipt)
       d    = (pt(1,ipt) - pt(1,jpt))**2 + (pt(2,ipt) - pt(2,jpt))**2  &
             + (pt(3,ipt) - pt(3,jpt))**2
       potl = potl - gsq * exp(-const*sqrt(d))/sqrt(d)
enddo
```

Indirect addressing example

```
ifort -O2 -xavx -qopt-report=3 -qopt-report-file=stderr  -qopt-report-phase= loop,vec -
qopt-report-routine=indirect drive_indirect.F90 indirect.F90;  ./a.out

LOOP BEGIN at indirect.F90(29,6)
    remark #15300: LOOP WAS VECTORIZED
…
    remark #15458: masked indexed (or gather) loads: 3
```

- The speedup compared to the non-vectorized version is ~2.7 x
  - This comes from vectorization of the computation, especially the exponential function, not from the indirect loads ("gathers") themselves.
  - If we compile with -xcore-avx2, hardware gather instructions are generated:

```
$ ifort -xcore-avx2 -S indirect.F90
$ grep gather indirect.s
    vgatherqps %xmm1, (,%ymm15), %xmm2
```

    – But the performance does not change

Indirect addressing

Whether vectorization is profitable depends on many factors, e.g.:

- Memory access pattern
- Amount of computation relative to memory access
- Whether loads are masked
- Loop trip count

You may sometimes want to override the compiler‘s choice:

!dir$ novector	(prevents vectorization of following loop)
!dir$ vector always	(overrides compiler's estimate of profitability)
!dir$ omp simd	(overrides everything)

The value of vectorizing gather (and sometimes scatter) operations is that it allows large loops to vectorize that otherwise would not

- the rest of the loop runs faster than if not vectorized
- The gather (or scatter) itself may not run faster

Scatter loops (b(ind(i)) = a(i) + …) introduce additional issues

- Different values of **i** may write to same element of **b**; possible race condition
- Interesting subject; Intel® AVX-512 conflict detection instructions help

More General Advice

Avoid manual unrolling in source (common in legacy codes)

- (re)write as simple “for” or “DO” loops
- Easier for the compiler to optimize and align
- Less platform-dependent
- More readable

Make loop induction variables local scalars (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

Disambiguate function arguments for C/C++

- E.g. By using -fargument-noalias

Explicit SIMD (Vector) Programming

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
- Directives are commands to the compiler, not hints
  - E.g. #pragma omp simd or !$OMP SIMD
  - Programmer is responsible for correctness (like OpenMP threading)
    - E.g. PRIVATE and REDUCTION clauses
  - Overrides all dependencies and cost-benefit analysis
- Now incorporated in OpenMP 4.0 $\Rightarrow$ portable
  - -qopenmp or -qopenmp-simd to enable

Explicit SIMD (Vector) Programming:

Use !$OMP SIMD or #pragma omp simd with -qopenmp-simd

```
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
```

loop was not vectorized:
existence of vector dependence.

```
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
```

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-pragma-simd/

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:

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

Example: Outer Loop Vectorization

```
    !  Calculate distance from data points to reference point
subroutine dist(pt, dis, n, nd, ptref)
  implicit none
  integer,                intent(in ) :: n, nd
  real, dimension(nd,n), intent(in ) :: pt
  real, dimension   (n), intent(out) :: dis
  real, dimension(nd),   intent(in ) :: ptref
  integer                             :: ipt, j
  real                                :: d

!$omp simd private(d)
  do ipt=1,n
     d = 0.

#ifdef KNOWN_TRIP_COUNT
     do j=1,MYDIM                !  Defaults to 3
#else
     do j=1,nd
#endif
        d = d + (pt(j,ipt) - ptref(j))**2
     enddo

     dis(ipt) = sqrt(d)
  enddo
 end
```

Outer loop with high trip count

Inner loop with low trip count

Outer Loop Vectorization

```
ifort -qopt-report-phase=loop,vec -qopt-report-file=stderr -c  dist.F90
…
LOOP BEGIN at dist.F90(17,3)
   remark #15542: loop was not vectorized: inner loop was already vectorized
…
 LOOP BEGIN at dist.F90(24,6)
     remark #15300: LOOP WAS VECTORIZED
```

We can vectorize the outer loop by activating the directive

`!$omp simd private(d)` using -qopenmp-simd

Each iteration must have its own "private" copy of d.

```
ifort -qopenmp-simd -qopt-report-phase=loop,vec -qopt-report-file=stderr
-qopt-report-routine=dist -c dist.F90
…
LOOP BEGIN at dist.F90(17,3)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  LOOP BEGIN at dist.F90(24,6)
     remark #25460: No loop optimizations reported
  LOOP END
```

Unrolling the Inner Loop

There is still an inner loop.
If the trip count is fixed and the compiler knows it,
the inner loop can be fully unrolled.

```
ifort -qopenmp-simd -DKNOWN_TRIP_COUNT -qopt-report-phase=loop,vec
-qopt-report-file=stderr -qopt-report-routine=dist drive_dist.F90 dist.F90
…
LOOP BEGIN at dist.F90(17,3)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

   LOOP BEGIN at dist.F90(22,6)
      remark #25436: completely unrolled by 3   (pre-vector)
   LOOP END
LOOP END
```

In this case, the outer loop can be vectorized more efficiently; SIMD may not be needed.

Outer Loop Vectorization - performance

Optimization Options	Speed-up	What's going on
-O1	1.0	No vectorization
-O2	1.1	Inner loop vectorization
-O2 -qopenmp-simd	1.7	Outer loop vectorization
-O2 -qopenmp-simd -DKNOWN_TRIP_COUNT	1.9	Inner loop fully unrolled
-O2 -qopenmp-simd -xcore-avx2 -DKNOWN_TRIP_COUNT	2.4	Intel® AVX2 including FMA instructions

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 beta.

Loops Containing Function Calls

Function calls can have side effects that introduce a loop-carried dependency, preventing vectorization

Possible remedies:

- Inlining
  - best for small functions
  - Must be in same source file, or else use -ipo
- !$OMP SIMD directive to vectorize remainder of loop, while preserving scalar calls to function (last resort)
- SIMD-enabled functions
  - Good for large, complex functions and in contexts where inlining is difficult
  - Call from regular DO loop
  - Adding “ELEMENTAL” keyword allows SIMD-enabled function to be called with array section argument

SIMD-enabled Function

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

```
     real function func(x,y,xp,yp)
!$omp declare simd (func) uniform( y, xp, yp )
        real, intent(in) :: x, y, xp, yp
         denom = (x-xp)**2 + (y-yp)**2
        func = 1./sqrt(denom)
     end
…
!$omp simd  private(x)  reduction(+:sumx)
        do i = 1,nx-1
          x = x0 + i*h
          sumx = sumx + func(x,y,xp,yp)
        enddo
```

y, xp and yp are constant, x can be a vector

FUNCTION WAS VECTORIZED with …

These clauses are required for correctness, just like for OpenMP*

SIMD LOOP WAS VECTORIZED.

SIMD-enabled function must have explicit interface

!$omp simd   may not be needed in simple cases

Clauses for SIMD-enabled Functions

#pragma omp declare simd	(C/C++)
!$OMP DECLARE SIMD (fn_name)	(Fortran)

• LINEAR (REF|VAL|UVAL)	(additional induction variables) use REF(X) when vector argument is passed by reference (Fortran default)
• UNIFORM	(argument is never vector)
• INBRANCH / NOTINBRANCH	(will function be called conditionally?)
• SIMDLEN	(vector length)
• ALIGNED	(tells compiler about data alignment)
• PROCESSOR	(tells compiler which processor to target. NOT controlled by –x… switch. Intel extension in 17.0 compiler)
• core_2nd_gen_avx	
• core_4th_gen_avx	
• mic_avx512, …	

Clauses for SIMD-enabled Functions

#pragma omp declare simd	(C/C++)
!$OMP DECLARE SIMD (fn_name)	(Fortran)

• LINEAR (REF|VAL|UVAL)	(additional induction variables) use REF(X) when vector argument is passed by reference (Fortran default)
• UNIFORM	(argument is never vector)
• INBRANCH / NOTINBRANCH	(will function be called conditionally?)
• SIMDLEN	(vector length)
• ALIGNED	(tells compiler about data alignment)
• PROCESSOR	(tells compiler which processor to
• core_2nd_gen_avx	target. NOT controlled by –x… switch.
• core_4th_gen_avx	Intel extension.)
• mic_avx512, …	(17.0 compiler only)

Use PROCESSOR clause to get full benefit on KNL

#pragma omp declare simd uniform(y,z,xp,yp,zp)

remark #15347: FUNCTION WAS VECTORIZED with **xmm**, simdlen=**4**, **unmasked**, formal parameter types: (vector,uniform,uniform,uniform)

remark #15347: FUNCTION WAS VECTORIZED with **xmm**, simdlen=**4**, **masked**, formal parameter types: (vector,uniform,uniform,uniform)

- default ABI requires passing arguments in 128 bit xmm registers

#pragma omp declare simd uniform(y,z,xp,yp,zp), **processor(mic-avx512), notinbranch**

remark #15347: FUNCTION WAS VECTORIZED with **zmm**, simdlen=**16**, **unmasked**, formal parameter types: (vector,uniform,uniform,uniform)

- Passing arguments in zmm registers facilitates 512 bit vectorization
- Independent of -xmic-avx512 switch
- notinbranch means compiler need not generate masked function version

SIMD-enabled Subroutine

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

```
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 test_linear
…
Interface
…
do j = 1,n
    call test_linear(a(j), b(j))
enddo
```

Important because arguments passed by reference in Fortran

remark #15301: FUNCTION WAS VECTORIZED.

remark #15300: LOOP WAS VECTORIZED.

SIMD-enabled routine must have explicit interface

!$omp simd   not needed in simple cases like this

SIMD-enabled Subroutine

The LINEAR(REF) clause is very important

- 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
  - New in compiler version 16.0.1
- Same method could be used for C arguments passed by reference

Approx speed-up for double precision array of 1M elements	
No DECLARE SIMD	1.0
DECLARE SIMD but no LINEAR(REF)	0.9
DECLARE SIMD with LINEAR(REF) clause	3.6

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.

Loop Optimization 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 to vectorize, if necessary
  - With compiler directives and hints
  - Using information from vectorization and optimization reports
  - With explicit vector programming
  - Use Intel® Advisor and/or Intel® VTune™ Amplifier XE to find the best places (hotspots) to focus your efforts
- No need to re-optimize vectorizable code for new processors
  - Typically a simple recompilation

Thread Safety

A threadsafe function can be called simultaneously from multiple threads, and still give correct results

- Potential conflicts must either be protected (synchronized) or avoided (by privatization)
- Static local data: each thread may access same copy!
- Automatic data: each thread has own copy & stack

ifort defaults:

- Local scalars are automatic
- Local arrays are static

When compiling with -openmp, default changes

- Local arrays become automatic
- Same as compiling with -auto
- This may increase the required stack size
  – Beware segmentation faults!

Making a function thread safe

With the compiler

- Compile with -qopenmp or -recursive
- Compile with -auto
  - May have less impact on serial optimization

In source code

- Use AUTOMATIC keyword in declarations
  - But doesn't cover compiler-generated temporaries
- Declare function as RECURSIVE
  - Covers whole function, including compiler-generated code
  - Best way to make code thread safe if you don't want to depend on build options

In either case:

- don't use -save or SAVE keyword
- Avoid global variables, (C also),
  - Or don't write to them unless synchronized

Thread Safe Libraries (Linux* Fortran)

The Intel® Math Kernel library is threadsafe

- Sequential version as well as threaded version

The Intel Fortran runtime library for Linux* has two versions

- The default is **not** threadsafe in compiler versions ≤ 16.0 (libifcore)
  - Build with -threads to link to threadsafe version (libifcoremt, libifcoremd)
- If you build with -qopenmp, the threadsafe version is linked by default
  - Compile main program with -qopenmp or -threads to ensure the threadsafe library is initialized in threadsafe mode.
  - If no Fortran main program, initialize library with

```
USE IFCORE
istat = FOR_SET_REENTRANCY(FOR_K_REENTRANCY_THREADED)
```

- Threadsafe version will be default in version 17.0 compiler

Performance considerations

Start with optimized serial code, vectorized inner loops, etc. (-O3 -xavx -ipo ...)

Ensure sufficient parallel work (/O3 /Qxavx /Qipo..)

Minimize data sharing between threads

- Unless read-only

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

```
!$OMP parallel do
 do i=1,nthreads
  do j=1,10000
    A(i,j) = A(i,j) + ..
```

Scheduling options

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

Timers for threaded apps

The Fortran standard timer CPU_TIME returns “processor time”

- It sums time over all threads/cores
- Like the “User time” in the Linux* “time” command
- So threaded apps don’t seem to run faster than serial ones☺

The Fortran intrinsic subroutine SYSTEM_CLOCK returns data from the real time clock

- Does not sum over cores
- Like the “real time” in the Linux “time” command
- Can be used to measure the speedup due to threading

dclock (Intel-specific function) can also be used

Thread Affinity Interface

Allows OpenMP* threads to be bound to physical or logical cores

- export environment variable OMP_PROC_BIND=
  - close assign threads to consecutive cores on same socket until saturated (eg to benefit from shared cache). May leave unused cores.
  - spread share threads between all cores but assign threads to consecutive cores on same socket where possible (maximize resources, including bandwidth to memory)
  - none (default) threads are not bound to cores
- Helps optimize access to memory or cache
- Particularly important if Hyperthreading is enabled
  - else some physical cores may be idle while others run multiple threads

- Older, Intel-specific variable is
  - KMP_AFFINITY = compact / scatter / balanced / physical
- See compiler documentation for more detail

Common problems

Insufficient stack size

- Most frequently reported OpenMP* issue!
- Typical symptom: seg fault during initialization

For whole program (shared + local data):

- Increase shell limits to large value
  - (address space, memory allocated only if needed)
- Bash : ulimit -s [value in KB] or [unlimited]
  - Can only increase once!
- C shell: limit stacksize 1000000 (for 1 GB)
- Windows*: /F:100000000 (value in bytes)
- Typical OS defaults ~10 MB

For individual thread (thread local data only)

- export **OMP_STACKSIZE**=[size], default 4m (4 MB)
- Actually allocates memory, so don’t make too large

Tips for Debugging OpenMP* apps

Build with -qopenmp to generate threaded code

- but run a single thread,   OMP_NUM_THREADS=1
- If it works, try Intel® Inspector XE to detect threading errors
- If still fails, excludes race conditions or other synchronization issues as cause

Build with -qopenmp-stubs -auto

- RTL calls are resolved; but no threaded code is generated
- allocates local arrays on the stack, as for OpenMP
- If works, check for missing FIRSTPRIVATE, LASTPRIVATE
- If still fails, eliminates threaded code generation as cause
- If works without -auto, implicates changed memory model
  - Perhaps insufficient stack size
  - Perhaps values not preserved between successive calls

Tips for Debugging OpenMP* apps

If debugging with PRINT statements

- print out the thread number with omp_get_thread_num()
- remember to USE OMP_LIB (declares this integer!)
- the internal I/O buffers are threadsafe (with -qopenmp),
  but the order of print statements from different threads is undetermined.

Debug with -O0 -qopenmp

- Unlike most other optimizations, OpenMP threading is not disabled at -O0

Heap or Stack?

If you don’t want to deal with stack size issues, you can ask for automatic and temporary arrays, including OpenMP* PRIVATE copies, to be allocated on the heap

-heap-arrays [size]

Arrays   [> size KB known at compile time] allocated on heap

ALLOCATE()   including inside the parallel region

Downside:

- Heap allocations will be serialized for safety
- Less data locality
- Some cost in performance, especially for high thread counts

Floating Point Programming Objectives (recap)

- **Accuracy**
  - Produce results that are "close" to the correct value
- **Performance**
  - Produce the most efficient code possible
- **Reproducibility**
  - Produce consistent or identical results
    - Beyond the inherent uncertainty of floating-point arithmetic
    - From one run to the next
    - From one set of build options to another (e.g. optimization levels)
    - From one compiler to another
    - From one platform to another
    - Typically for reasons related to QA

These objectives usually conflict!
Compiler options let you control the tradeoffs.

Original Solution: /fp:precise (-fp-model precise)

For consistent results from non-threaded, compiler-generated code

- Based on Microsoft* Visual C++ option
- Introduced in Intel® Compiler version 9.
- Allows value-safe optimizations only. Examples of optimizations disallowed:
  - Reassociation
    - Includes vectorization of reductions
  - Flush-to zero (abrupt underflow)
  - Fast, approximate divide and sqrt
  - Fast, vectorized math functions
  - Constant folding (evaluation at compile time)
  - Etc.

But things got more complicated…

Math libraries gave different results on different microarchitectures

- Including on Intel and non-Intel processors
- Due to internal dispatch inside library at run-time
  - Independent of compiler processor-targeting switches

Solution: `/Qimf-arch-consistency:true` (`-fimf-arch-consistency=true`)

- Gives same result on all processors of same architecture
  - At some cost in performance

And then there were FMAs …

An FMA may give a different result from separate multiply and add operations

- Normally, more accurate
- Newly important since the arrival of Intel® Xeon Phi™ coprocessors and processors supporting Intel® AVX2
  - Though Intel® Itanium® processors had them, too
- FMA's (F-P contractions) are not disabled by /fp:precise (-fp-model precise)
  - Original decision by Microsoft
  - Leads to differences between processors that do or do not support FMAs
- Replacing a multiply and an add by an FMA is an optimization
  - At -O1 and above, with -xcore-avx2, -mmic, -xmic-avx512, etc.
  - Leads to different results at different optimization levels
  - Not covered by language standards; compiler may choose to group operations into FMAs in different ways

FMA: Example and Solution

FMA's break symmetry, e.g.:

```
double a, b, c, d, x;
c = -a;
d =  b;
x = a*b + c*d;
```

- Without FMA, should evaluate to zero; with FMA, may not:

```
x =  (a* b + (c*d))    or    ((a*b) + c * d)
     FMA(a,b,(c*d))          FMA(c,d,(a*b))
```

  - each has different rounding, unspecified which grouping the compiler will generate

Solution for reproducible results is /Qfma- (-no-fma)

- At some cost in performance
  - /fp:strict (-fp-model strict) also disables, but has much wider impact…

Reproducibility: the bottom line (for Intel64)

/fp:precise /Qfma- /Qimf-arch-consistency:true (Windows*)

-fp-model precise -no-fma -fimf-arch-consistency=true (Linux* or OS X*)

- Recommended for best reproducibility
  - Also for IEEE compliance
  - And for language standards compliance (C, C++ and Fortran)

- This isn't very intuitive
  - a single switch will do all this in the 17.0 compiler
  - -fp-model consistent (/fp:consistent on Windows*)

Submodules from Fortran 2008

Minor, internal changes to a module no longer trigger recompilation of all code that USEs the module

- Provided interfaces do not change

Interfaces go in parent module

- And any variables that are accessed externally

Implementations go in submodule(s)

```
SUBMODULE  (My_Mod)  My_Sub_Mod
```

- Separate source
- Variables cannot be accessed outside of submodule
- Need not repeat interface if MODULE procedure

New intermediate files *.smod (like .mod file for submodules)

Submodules (F2008) – The Solution

```
module bigmod
…
interface
module subroutine sub1
…
module function func2
…
module subroutine sub47
…
end interface
end module bigmod
```

```
submodule (bigmod)
bigmod_submod
contains
module subroutine sub1
… implementation of sub1
module function func2
… implementation of func2
…
module subroutine sub3
… implementation of sub3
end submodule bigmod_submod
```

Changes in the submodule don't force recompilation of uses of the module – as long as the interface doesn't change

Scalar Math Library

Scalar math library now optimized for Intel® AVX2

- libimf on Linux* and OS*X, libm on Windows*
- FMA instructions, in particular, lead to speed up
  - Somewhat more for double precision than for single
  - tan, sin, cos, exp, pow…
- Intel® AVX2 support detected at run-time and corresponding function version is selected
- There are no scalar math library special optimizations for Intel® AVX, since the increased vector width does not directly benefit scalar code.
- The Short Vector Math Library (libsvml) contains vectorized function versions with their own entry points for different instruction sets, optimized for Intel AVX2 since compiler version 13.0.

More New Features in Fortran 16.0

IMPURE ELEMENTAL from Fortran 2008

- ELEMENTAL procedures can have side effects, e.g. I/O, random numbers,

Further C Interoperability from Fortran 2015

- TYPE(*) (assumed type, rather like void* in C)
- DIMENSION(..) (assumed rank)
- Support needs of MPI3

!$OMP TARGET DEPEND (OpenMP* 4.5)

- Facilitates synchronization between offload tasks

!DIR$ BLOCK_LOOP

- Facilitates blocking for better data reuse and cache utilization

-fpp-name option

- Lets you supply your own preprocessor

New Fortran Features of 17.0

Much improved coarray performance

- No more element-by-element transfers

Default RTL is threadsafe at last!

- -reentrancy=threaded

Assert alignment of allocatable arrays (and pointer arrays)

- Didn't work before 17.0
  - Compiler can't know lower bound at compile time
- !dir$ assume_aligned   ARRAY(-4):64
  - Must specify which element is aligned!

Less temporary array copies:

- CONTIGUOUS keyword may require temporary copy to ensure contiguity
  - Without CONTIGUOUS, assumed shape arrays result in gathers
- 17.0 compiler recognizes more contiguous arrays at compile time
  - CQ409519 – allocatable components of derived types
- tests for contiguity at run-time. (CQ380398)

- %re, %im   for real & imaginary parts of complex number

New OpenMP Features of 17.0

LINEAR(REF( )) clause for SIMD functions

- Avoids gathers for pass-by-reference => better performance
- Particularly important for Fortran, where pass by reference is default

```
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 test_linear
```

PROCESSOR clause for SIMD functions

- Else ABI requires arguments passed in 128 bit xmm registers, even on KNL
- !$OMP DECLARE SIMD (my_func) PROCESSOR (mic_avx512)   (or #pragma)

TASKLOOP

- Somewhat similar to cilk_for; thread a loop using OpenMP tasks

New C/C++ Features of 17.0

- SIMD Data Layout Template to facilitate vectorization for your C++ code
  - Write code as AoS, let compiler convert to SoA for optimization
- Improved code alignment for loops and functions
  - -falign-loops[=n], /Qalign-loops[:n]
- support for most of the latest C11 and C++14 standards
  - C11 keywords _Alignas, _Alignof, _Static_assert, _Thread_local, _Noreturn, _Generic
  - C++14: Generic Lambdas, Generalized lambda captures, Digit Separators, [[deprecated]] attribute
- Virtual Vector Functions: Set of versions inherited and cannot be altered in overrides
- Vector Function Pointers: -simd-function-pointers, -Qsimd-function-pointers
- SVRNG (Short Vector Random Number Generator): SIMD intrinsics for high quality random number generators from the C++ standard and/or Intel® MKL
  - rand0, rand, mcg31m1, mcg59, mt19937
  - Enables vectorization of loops with scalar calls to these functions

KNL Overview

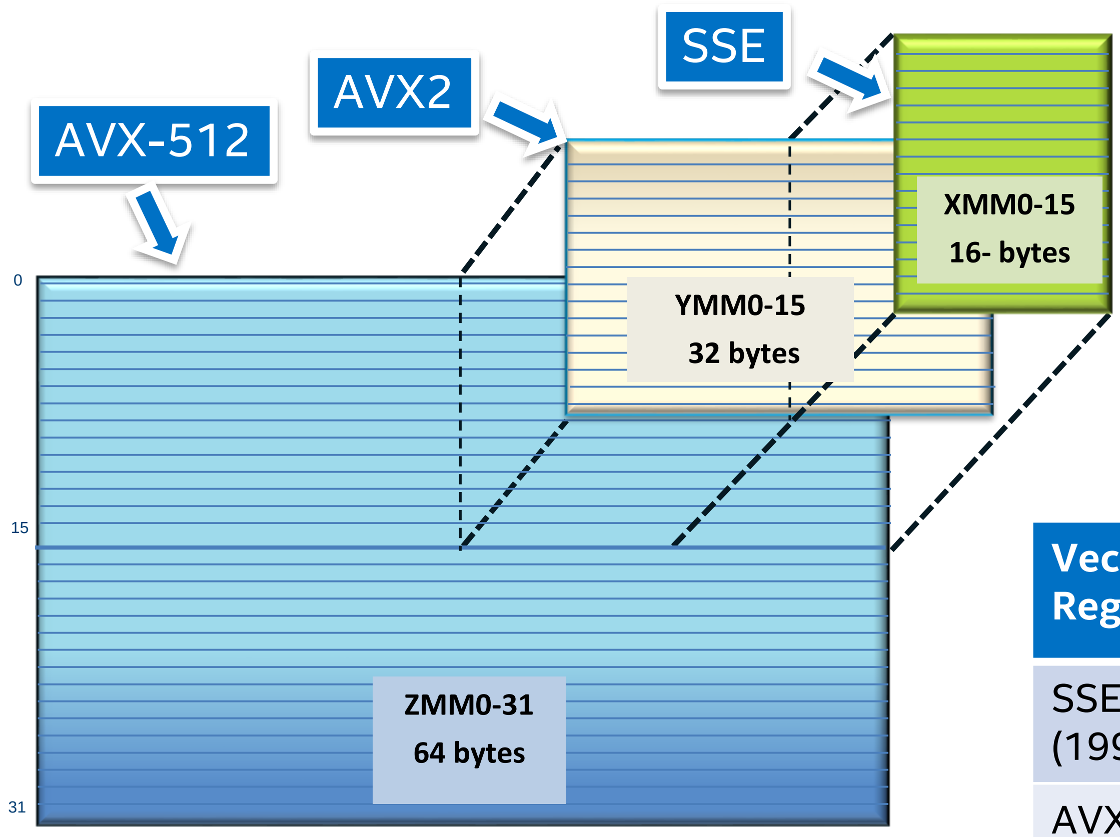
- Self- or leveraged-boot
  - But expect most customers to prefer self-boot (no more offload!)
- Intel® AVX-512 instruction set
  - Slightly different from future Intel® Xeon architecture

    Binary incompatible with KNC (mostly source compatible)
- Intel® SSE, AVX, AVX2 instruction sets
  - Apps built for HSW and earlier can run on KNL without recompilation
- More cores than KNC, higher frequency
  - Silvermont-based, better scalar performance
- New, on-package high bandwidth memory (MCDRAM)
- Lots of regular memory (100's of GB DDR4)
  - Run much larger HPC workloads than KNC

KNL INSTRUCTION SET ARCHITECTURE

Changes for SW development resulting from new Intel® AVX-512 ISA

AVX-512 - Greatly increased Register File



Vector Registers	IA32 (32bit)	Intel64 (64bit)
SSE (1999)	8 x 128bit	16 x 128bit
AVX and AVX-2 (2011 / 2013)	8 x 256bit	16 x 256bit
AVX-512 (2014 – KNL)	8 x 512bit	32 x 512bit

The Intel® AVX-512 Subsets [1]

AVX-512 F

AVX-512 F: 512-bit Foundation instructions common between MIC and Xeon

- ❑ Comprehensive vector extension for HPC and enterprise
- ❑ All the key AVX-512 features: masking, broadcast…
- ❑ 32-bit and 64-bit integer and floating-point instructions
- ❑ Promotion of many AVX and AVX2 instructions to AVX-512
- ❑ Many new instructions added to accelerate HPC workloads

AVX-512CD

AVX-512 CD (Conflict Detection instructions)

- ❑ Allow vectorization of loops with possible address conflict
- ❑ Will show up on Xeon

AVX-512ER

AVX-512PR

AVX-512 extensions for exponential and prefetch operations

- ❑ fast (28 bit) instructions for exponential and reciprocal (as well as RSQRT)
- ❑ New prefetch instructions: gather/scatter prefetches and PREFETCHWT1

The Intel® AVX-512 Subsets [2] (not KNL !)

AVX-512DQ

AVX-512 Double and Quad word instructions

- ❑ All of (packed) 32bit/64 bit operations AVX-512F doesn't provide
- ❑ Close 64bit gaps like VPMULLQ : packed 64x64 ➔ 64
- ❑ Extend mask architecture to word and byte (to handle vectors)
- ❑ Packed/Scalar converts of signed/unsigned to SP/DP

AVX-512BW

AVX-512 Byte and Word instructions

- ❑ Extend packed (vector) instructions to byte and word (16 and 8 bit) data types
  - ❑MMX/SSE2/AVX2 re-promoted to AVX512 semantics
- ❑ Mask operations extended to 32/64 bits to adapt to number of objects in 512bit
- ❑ Permute architecture extended to words (VPERMW, VPERMI2W, …)

AVX-512VL

AVX-512 Vector Length extensions

- ❑ Vector length orthogonality
  - ❑Support for 128 and 256 bits instead of full 512 bit
- ❑ Not a new instruction set but an attribute of existing 512bit instructions

Other New Instructions (not KNL!)

MPX

Intel® MPX – Intel Memory Protection Extension

- ❑Set of instructions to implement checking a pointer against its bounds
- ❑Pointer Checker support in HW (today a SW only solution of e.g. Intel Compilers)
- ❑Debug and security features

SHA

Intel® SHA – Intel Secure Hash Algorithm

- ❑ Fast implementation of cryptographic hashing algorithm as defined by NIST FIPS PUB 180

CLFLUSHOPT

Single Instruction – Flush a cache line

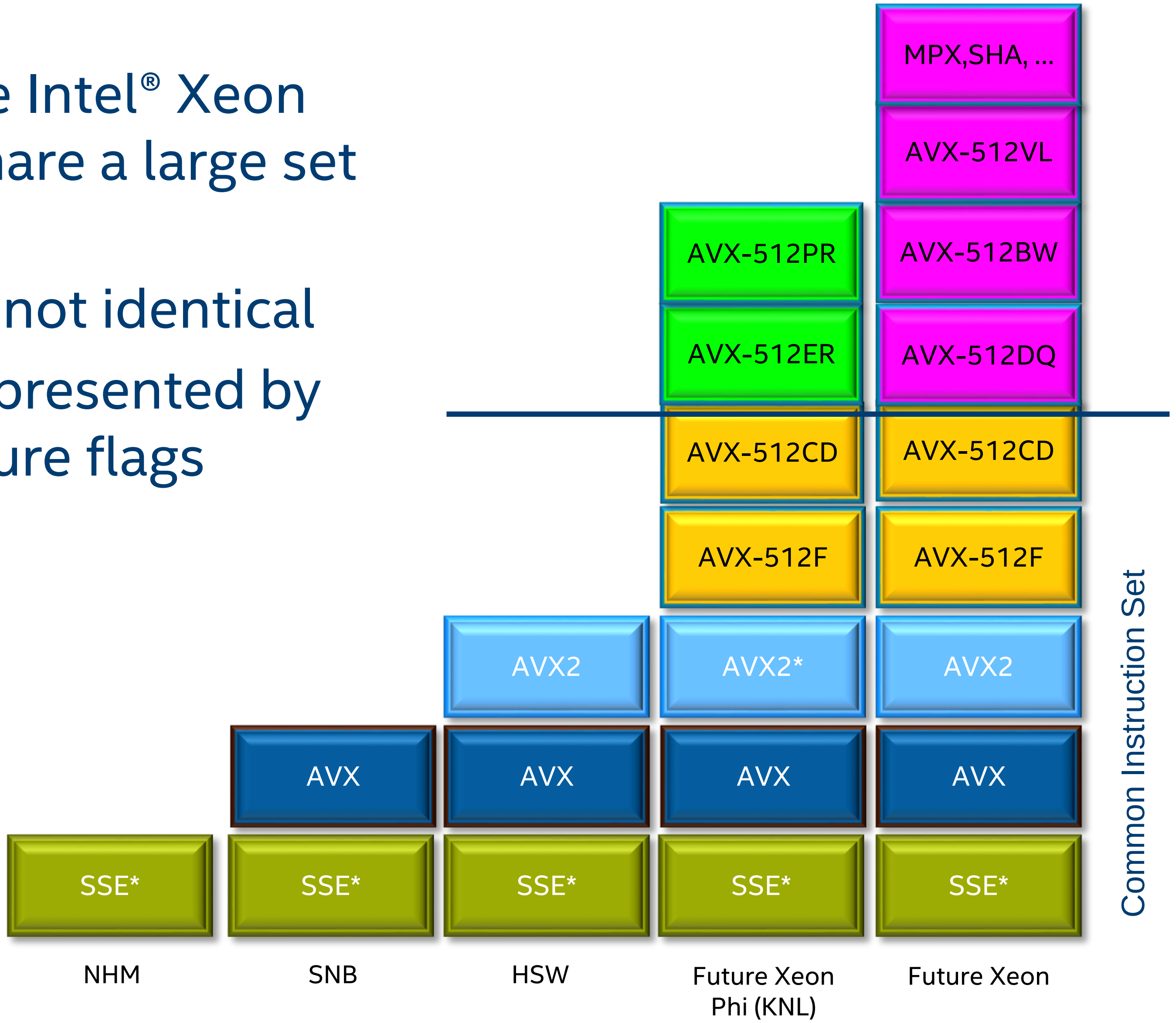
- ❑ needed for future memory technologies

XSAVE{S,C}

Save and restore extended processor state

AVX-512 – KNL and future Xeon

- KNL and future Intel® Xeon architecture share a large set of instructions
  - but sets are not identical
- Subsets are represented by individual feature flags (CPUID)



Intel® Compiler Switches Targeting Intel® AVX-512

Switch	Description
`-xmic-avx512`	KNL only
`-xcore-avx512`	Future Xeon only
`-xcommon-avx512`	AVX-512 subset common to both. <u>Not</u> a fat binary.
`-m, -march, /arch`	Not yet !
`-axmic-avx512 etc.`	Fat binaries. Allows to target KNL and other Intel® Xeon® processors
`-qoffload-arch=mic-avx512`	Offload to KNL coprocessor

Don't use -mmic with KNL !

All supported in 16.0 and 17.0 compilers

Binaries built for earlier Intel® Xeon® processors will run unchanged on KNL
Binaries built for Intel® Xeon Phi™ coprocessors will not.

Consider Cross-Compiling

KNL is suited to highly parallel applications

- It's scalar processor is less powerful than that of a "large core" Intel® Xeon® processor

The Intel® Compiler is a mostly serial application

- Compilation is likely to be faster on an Intel Xeon processor
- For parallelism, try make -j

Improved Optimization Report

```
subroutine test1(a, b ,c, d)
  integer, parameter        :: len=1024
  complex(8), dimension(len) :: a, b, c
  real(4),    dimension(len) :: d
  do i=1,len
     c(i) = exp(d(i)) +  a(i)/b(i)
  enddo
End
```

From assembly listing:

```
# VECTOR LENGTH 16
# MAIN VECTOR TYPE: 32-bits floating point
```

```
$ ifort -c -S -xmic-avx512  -O3  -qopt-report=4  -qopt-report-file=stderr
-qopt-report-phase=loop,vec,cg  -qopt-report-embed  test_rpt.f90
```

- 1 vector iteration comprises
  - 16 floats in a single AVX-512 register (d)
  - 16 double complex in 4 AVX-512 registers per variable (a, b, c)

- Replace exp(d(i)) by d(i) and the compiler will choose a vector length of 4
  - More efficient to convert d immediately to double complex

Improved Optimization Report

```
Compiler options: -c -S -xmic-avx512 -O3 -qopt-report=4 -qopt-report-file=stderr
-qopt-report-phase=loop,vec,cg -qopt-report-embed
…
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized vectorization overhead 0.087
   remark #15417: vectorization support: number of FP up converts: single
                  precision to double precision 1   [ test_rpt.f90(7,6) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15482: vectorized math library calls: 1
   remark #15486: divides: 1
   remark #15487: type converts: 1
…
```

- New features include the code generation (CG) / register allocation report
  - Includes temporaries; stack variables; spills to/from memory

Optimization Improvements

Vectorization works as for other targets

- 512, 256 and 128 bit instructions available
- 64 byte alignment is best, like for KNC
- New instructions can help

Vectorization of compress/expand loops:

- Uses vcompress/vexpand on KNL

```
for (int i; i <N; i++)  {
    if (a[i] > 0)   {
        b[j++]  = a[i];   //
compress
        c[i]  = a[k++];  //
expand
    }
}
```

- Cross-iteration dependencies by j and k

Convert certain gathers to vector loads

Can auto-generate Conflict Detection instructions (AVX512CD)

Compress/Expand Loops with Intel® AVX-512

```
nb = 0
 do ia=1, na                ! line 11
  if(a(ia) > 0.) then
   nb = nb + 1
   b(nb) = a(ia)
  endif
 enddo
```

```
for (int i; i <N; i++)  {
    if (a[i] > 0)   {
        b[j++]  = a[i];   // compress
//        c[i]  = a[k++];  // expand
    }
}
```

- Cross-iteration dependencies by j and k

With Intel® AVX2, does not auto-vectorize

- And vectorizing with SIMD would be too inefficient

```
ifort -c -xcore-avx2 -qopt-report-file=stderr -qopt-report=3 -qopt-report-phase=vec compress.f90
 …
 LOOP BEGIN at compress.f90(11,3)
   remark #15344: loop was not vectorized: vector dependence prevents vectorization.
                            First dependence is shown below. Use level 5 report for details
   remark #15346: vector dependence: assumed ANTI dependence between  line 13 and  line 13
 LOOP END
```

- C code behaves the same

Compress Loop

Compile for KNL:

```
ifort -c -qopt-report-file=stderr -qopt-report=3 -qopt-report-phase=vec -xmic-avx512 compress.f90
…
LOOP BEGIN at compress.f90(11,3)
  remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 1
  remark #15457: masked unaligned unit stride stores: 1
…
  remark #15478: estimated potential speedup: 14.040
  remark #15497: vector compress: 1
LOOP END

grep vcompress compress.s
      vcompressps %zmm4, -4(%rsi,%rdx,4){%k1}         #14.7 c7 stall 1
      vcompressps %zmm1, -4(%rsi,%r12,4){%k1}         #14.7 c5
      vcompressps %zmm1, -4(%rsi,%r12,4){%k1}         #14.7 c5
      vcompressps %zmm4, -4(%rsi,%rdi,4){%k1}         #14.7 c7 stall 1
```

Observed speed-up is substantial but depends on problem size, data layout, etc.

Adjacent Gather Optimizations

Or “Neighborhood Gather Optimizations”

```
do j=1,n
    y(j) = x(1,j) + x(2,j) + x(3,j) + x(4,j) ....
```

- Elements of x are adjacent in memory, but vector index is in other dimension
- Compiler generates simd loads and shuffles for x instead of gathers
  - Before AVX-512: gather of x(1,1), x(1,2), x(1,3), x(1,4)
  - With AVX-512: SIMD loads of x(1,1), x(2,1), x(3,1), x(4,1) etc., followed by permutes to get back to x(1,1), x(1,2), x(1,3), x(1,4) etc.
  - Message in optimization report:

    remark #34029: adjacent sparse (indexed) loads optimized for speed
- Arrays of short vectors or structs are very common

Motivation for Conflict Detection

Sparse computations are common in HPC, but hard to vectorize due to race conditions

Consider the “scatter” or “histogram” problem:

```
for(i=0; i<16; i++) {  A[B[i]]++; }
```

```
index = vload &B[i]                 // Load 16 B[i]
old_val = vgather A, index          // Grab A[B[i]]
new_val = vadd old_val, +1.0        // Compute new values
vscatter A, index, new_val          // Update A[B[i]]
```

- Problem if two vector lanes try to increment the same histogram bin
- Code above is wrong if any values within B[i] are duplicated
  - Only one update from the repeated index would be registered!
- A solution to the problem would be to avoid executing the sequence gather-op-scatter with vector of indexes that contain conflicts

Conflict Detection Instructions in AVX-512

VPCONFLICT instruction detects elements with previous conflicts in a vector of indexes

- Allows to generate a mask with a subset of elements that are guaranteed to be conflict free
- The computation loop can be re-executed with the remaining elements until all the indexes have been operated upon

VCONFLICT instr.
VPCONFLICT{D,Q} zmm1{k1}, zmm2/mem
VPBROADCASTM{W2D,B2Q} zmm1, k2
VPTESTNM{D,Q} k2{k1}, zmm2, zmm3/mem
VPLZCNT{D,Q} zmm1 {k1}, zmm2/mem

```
index = vload &B[i]                                  // Load 16 B[i]
pending_elem = 0xFFFF;                               // all still remaining
do {
   curr_elem = get_conflict_free_subset(index, pending_elem)
   old_val = vgather {curr_elem} A, index            // Grab A[B[i]]
   new_val = vadd old_val, +1.0                      // Compute new values
   vscatter A {curr_elem}, index, new_val            // Update A[B[i]]
   pending_elem = pending_elem ^ curr_elem           // remove done idx
} while (pending_elem)
```

Histogramming with Intel® AVX2

```
!     Accumulate histogram of sin(x) in  h
      do i=1,n
          y     = sin(x(i)*twopi)
          ih    = ceiling((y-bot)*invbinw)
          ih    = min(nbin,max(1,ih))
          h(ih) = h(ih) + 1                  ! line 15
      enddo
```

With Intel® AVX2, this does not vectorize

- Store to **h** is a scatter
- **ih** can have the same value for different values of **i**
- Vectorization with a SIMD directive would cause incorrect results

```
ifort -c -xcore-avx2 histo2.f90 -qopt-report-file=stderr -qopt-report-phase=vec
LOOP BEGIN at histo2.f90(11,4)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization.
                 First dependence is shown below. Use level 5 report for details
  remark #15346: vector dependence: assumed FLOW dependence between  line 15 and  line 15
LOOP END
```

Histogramming with Intel® AVX-512

Compile for KNL using Intel® AVX-512CD:

```
ifort -c -xmic-avx512 histo2.f90 -qopt-report-file=stderr -qopt-report=3 –S
...
LOOP BEGIN at histo2.f90(11,4)
   remark #15300: LOOP WAS VECTORIZED
   remark #15458: masked indexed (or gather) loads: 1
   remark #15459: masked indexed (or scatter) stores: 1
   remark #15478: estimated potential speedup: 13.930
   remark #15499: histogram: 2
LOOP END
```

Some remarks omitted

```
vpminsd      %zmm5, %zmm21, %zmm3                 #14.7 c19
vpconflictd  %zmm3, %zmm1                         #15.7 c21
vpgatherdd   -4(%rsi,%zmm3,4), %zmm6{%k1}         #15.15 c21
vptestmd     %zmm18, %zmm1, %k0                   #15.7 c23
kmovw        %k0, %r10d                           #15.7 c27 stall 1
vpaddd       %zmm19, %zmm6, %zmm2                 #15.7 c27
testl        %r10d, %r10d
...
 vpscatterdd %zmm2, -4(%rsi,%zmm3,4){%k1}         #15.7 c3
```

Histogramming with Intel® AVX-512

Observed speed-up between AVX2 (non-vectorized) and AVX512 (vectorized) can be large, but depends on problem details

- Comes mostly from vectorization of other heavy computation in the loop
  - Not from the scatter itself
- Speed-up may be (much) less if there are many conflicts
  - E.g. histograms with a singularity or narrow spike

Other problems map to this

- E.g. energy deposition in cells in particle transport Monte Carlos

Prefetching for KNL

Hardware prefetcher is more effective than for KNC

Software (compiler-generated) prefetching is off by default

- Like for Intel® Xeon® processors
- Enable by -qopt-prefetch=[1-5]

KNL has gather/scatter prefetch

- Enable auto-generation to L2 with -qopt-prefetch=5
  - Along with all other types of prefetch, in addition to h/w prefetcher – careful.
- Or hint for specific prefetches
  - !DIR$ PREFETCH var_name [: type : distance]
  - Needs at least -qopt-prefetch=2
- Or call intrinsic
  - **`_mm_prefetch((char *) &a[i], hint);`**
  - **`MM_PREFETCH(A, hint)`**

Prefetching for KNL

```
void foo(int n, int* A, int *B, int *C)   {
    // pragma_prefetch var:hint:distance
#pragma prefetch A:1:3
#pragma vector aligned
#pragma simd
  for(int i=0; i<n; i++)
    C[i] = A[B[i]];
}
```

icc **-O3 -xmic-avx512 -qopt-prefetch=3** -qopt-report=4 -qopt-report-file=stderr -c -S emre5.cpp

```
   remark #25033: Number of indirect prefetches=1, dist=2
   remark #25035: Number of pointer data prefetches=2, dist=8
   remark #25150: Using directive-based hint=1, distance=3 for indirect memory reference  [ emre5.cpp(...
   remark #25540: Using gather/scatter prefetch for indirect memory reference, dist=3   [ emre5.cpp(9,12) ]
   remark #25143: Inserting bound-check around lfetches for loop
```

% grep gatherpf emre5.s

```
        vgatherpf1dps (%rsi,%zmm0){%k1}                   #9.12 c7 stall 2
```

% grep prefetch emre5.s

```
# mark_description "-O3 -xmic-avx512 -qopt-prefetch=3 -qopt-report=4 -qopt-report-file=stderr -c -S -g";
        prefetcht0 512(%r9,%rcx)                          #9.14 c1
        prefetcht0 512(%r9,%r8)                           #9.5 c7
```

Intel® Software Development Emulator (SDE)

Intel Compilers Already Recognize Intel® AVX-512 and Will Generate KNL Code

Use Intel® Software Development Emulator (SDE) to Test Code

- Will test instruction mix, not performance
- Does not emulate hardware (e.g. memory hierarchy) only ISA

Use the SDE to Answer

- Is my compiler generating Intel® AVX-512/KNL-ready code?
- How do I restructure my code so that Intel® AVX-512 code is generated?

Visit *Intel Xeon Phi Coprocessor code named "Knights Landing" - Application Readiness*

KNL HIGH BANDWIDTH MEMORY

Adapting SW to make best use of KNL MCDRAM

High Bandwidth On-Package Memory API

- API is open-sourced (BSD licenses)
  - https://github.com/memkind ; also part of XPPSL at https://software.intel.com/articles/xeon-phi-software
  - User jemalloc API underneath
    - http://www.canonware.com/jemalloc/
    - https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919

malloc replacement:

```
#include <memkind.h>

  hbw_check_available()
  hbw_malloc, _calloc, _realloc,… (memkind_t kind, …)
  hbw_free()
  hbw_posix_memalign(), _posix_memalign_psize()
  hbw_get_policy(), _set_policy()

ld …  -ljemalloc –lnuma –lmemkind –lpthread
```

HBW API for Fortran, C++

Fortran:

!DIR$ ATTRIBUTES FASTMEM :: data_object1, data_object2   (15.0, 16.0, 17.0)

- Flat or hybrid mode only
- More Fortran data types may be supported eventually
  - Global, local, stack or heap;   OpenMP private copies;
  - Currently just allocatable arrays (16.0) and pointers (17.0)
  - Must remember to link with libmemkind !

Possible addition in a 17.0 compiler:

- Placing FASTMEM directive before ALLOCATE statement
  - Instead of ALLOCATABLE declaration

C++:   can pass hbw_malloc()   etc.

standard allocator replacement for e.g. STL like

```
#include <hbw_allocator.h>
std::vector<int, hbw::allocator::allocate>
```

Available already, working on documentation

HBW APIs (Fortran)

Use Fortran 2003 C-interoperability features to call memkind API

```
interface
  function hbw_check_available() result(avail) bind(C,name='hbw_check_available')
    use iso_c_binding
    implicit none
    integer(C_INT) :: avail
  end function hbw_check_available
end interface

integer :: istat
istat = hbw_check_available()
if (istat == 0) then
    print *, HBM available'
  else
    print *, 'ERROR, HBM not available, return code=', istat
end if
```

How much HBM is left?

```
#include <memkind.h>
int hbw_get_size(int partition, size_t * total, size_t * free) {          // partition=1   for HBM
  memkind_t kind;

  int stat = memkind_get_kind_by_partition(partition, &kind);
  if(stat==0) stat = memkind_get_size(kind, total, free);
  return stat;
}
```

Fortran interface:

```
 interface
   function hbw_get_size(partition, total, free) result(istat)  bind(C, name='hbw_get_size')
     use iso_c_binding
     implicit none
     integer(C_INT)           :: istat
     integer(C_INT), value :: partition
     integer(C_SIZE_T)       :: total, free
   end function hbw_get_size
  end interface
```

HBM doesn't show as "used" until first access after allocation

What Happens if HBW Memory is Unavailable? (Fortran)

In 16.0: silently default over to regular memory

New Fortran intrinsic in module IFCORE in 17.0:

integer(4) FOR_GET_HBW_AVAILABILITY()

Return values:

- FOR_K_HBW_NOT_INITIALIZED (= 0)
  - Automatically triggers initialization of internal variables
  - In this case, call a second time to determine availability
- FOR_K_HBW_AVAILABLE (= 1)
- FOR_K_HBW_NO_ROUTINES (= 2) e.g. because libmemkind not linked
- FOR_K_HBW_NOT_AVAILABLE (= 3)
  - does not distinguish between HBW memory not present; too little HBW available; and failure to set MEMKIND_HBW_NODES

New RTL diagnostics when ALLOCATE to fast memory cannot be honored:

183/4 warning/error libmemkind not linked
185/6 warning/error HBW memory not available
Severe errors 184, 186 may be returned in STAT field of ALLOCATE statement

Controlling What Happens if HBM is Unavailable (Fortran)

In 16.0: you can't

New Fortran intrinsic in module IFCORE in 17.0:

integer(4) FOR_SET_FASTMEM_POLICY(new_policy)

input arguments:

FOR_FASTMEM_INFO	(= 0)	return current policy unchanged
FOR_FASTMEM_NORETRY	(= 1)	error if unavailable (**default**)
FOR_FASTMEM_RETRY_WARN	(= 2)	warn if unavailable, use default memory
FOR_FASTMEM_RETRY	(= 3)	if unavailable, silently use default memory

- returns <u>previous</u> HBW policy

Environment variables (to be set before program execution):

FOR_FASTMEM_NORETRY	=T/F	default False
FOR_FASTMEM_RETRY	=T/F	default False
FOR_FASTMEM_RETRY_WARM	=T/F	default False

KNL FLOATING POINT CONSISTENCY

Dealing with FP consistency when moving from Intel® MIC or Intel Xeon to KNL

Floating-Point Reproducibility

-fp-model precise   disables most value-unsafe optimizations (especially reassociations)

- The primary way to get consistency between different platforms (including KNL) or different optimization levels
- Does not prevent differences due to:
  - Different implementations of math functions
  - Use of fused multiply-add instructions (FMAs)

- Floating-point results on Intel® Xeon Phi™ coprocessors may not be bit-for-bit identical to results obtained on Intel® Xeon® processors or on KNL

Disabled by -fp-model precise

Vectorization of loops containing transcendental functions

Fast, approximate division and square roots

Flush-to-zero of denormals

Vectorization of reduction loops

Other reassociations

  (including hoisting invariant expressions out of loops)

Evaluation of constant expressions at compile time

…

Additional Resources (Optimization)

Webinars:
https://software.intel.com/videos/getting-the-most-out-of-the-intel-compiler-with-new-optimization-reports
https://software.intel.com/videos/new-vectorization-features-of-the-intel-compiler
https://software.intel.com/articles/further-vectorization-features-of-the-intel-compiler-webinar-code-samples
https://software.intel.com/videos/from-serial-to-awesome-part-2-advanced-code-vectorization-and-optimization
https://software.intel.com/videos/data-alignment-padding-and-peel-remainder-loops

Vectorization Guide (C): https://software.intel.com/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/

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

https://software.intel.com/articles/fortran-array-data-and-arguments-and-vectorization

Compiler User Forums at http://software.intel.com/forums

Some Reproducibility References

“Consistency of Floating-Point Results using the Intel® Compiler”
http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/

“Differences in Floating-Point Arithmetic between Intel® Xeon® Processors and the Intel® Xeon Phi™ Coprocessor”
http://software.intel.com/sites/default/files/article/326703/floating-point-differences-sept11.pdf

“Run-to-Run Reproducibility of Floating-Point Calculations for Applications on Intel® Xeon Phi™ Coprocessors (and Intel® Xeon® Processors)”
https://software.intel.com/en-us/articles/run-to-run-reproducibility-of-floating-point-calculations-for-applications-on-intel-xeon

Intel® Compiler User and Reference Guides:
https://software.intel.com/intel-cplusplus-compiler-16.0-user-and-reference-guide
https://software.intel.com/intel-fortran-compiler-16.0-user-and-reference-guide
 “Floating Point Operations”

Summary

Intel provides a powerful, optimizing compiler for x86 architecture and for Intel® MIC architecture

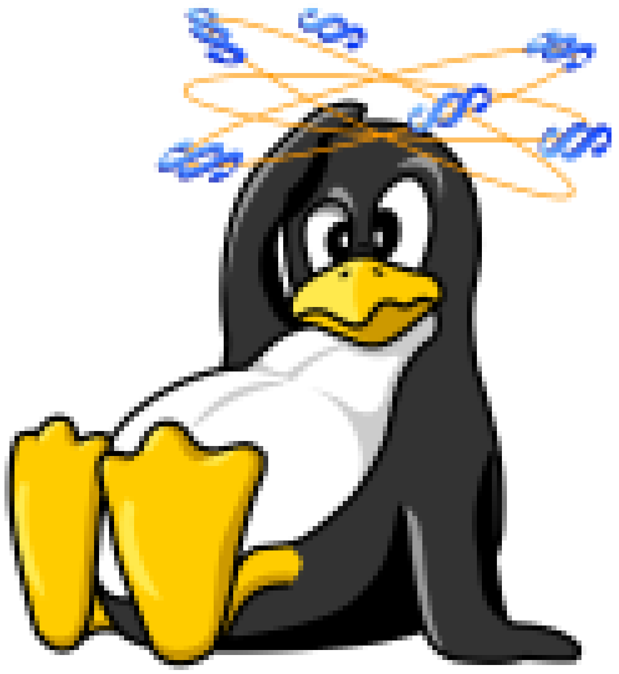
- Best performance on Intel architecture, and competitive performance on non-Intel systems
- More optimizations in the pipeline

Our focus is on

- Performance
- Comprehensive coverage of parallelism
- Ease of use
- Compatibility and software investment protection
- Customer Support

Visit http://software.intel.com/developer-tools-technical-enterprise

Questions?



Legal Disclaimer & Optimization Notice

BACKUP

Including C versions of some Fortran slides and Fortran versions of some C slides

Also vectorization of STL vectors and valarrays

Compatibility

Linux*:

Intel® C and C++ are binary compatible with gcc, g++, gdb, …

- Substantially source and command line compatible

Intel® Fortran is also binary compatible with gcc, gdb, …

- But <u>not</u> binary compatible with gfortran
- Facilitated by Fortran 2003 C interoperability features

GCC compatibility and interoperability: see the 16.0 user guide

- https://software.intel.com/node/583823

Compatibility

Windows*:

Intel® C and C++ are binary compatible with Microsoft* Visual C++

- Highly source and command line compatible

Intel® Fortran is also binary compatible with Microsoft* Visual C++

- Also supports a variety of Windows APIs
- Compiler doesn't support C# directly
  - But C++ and Fortran DLLs can be called from C#   (example)
  - Passing strings to/from Fortran can be tricky (see user guide)
    - See :mixed language programming" in the Fortran user guide

Which Microsoft* Visual Studio* versions can I use with Intel® Compilers ?

- https://software.intel.com/intel-parallel-studio-xe-compilers-required-microsoft-visual-studio
- https://software.intel.com/articles/troubleshooting-fortran-integration-issues-with-visual-studio

Optimization Overview

The days of easy performance improvements from steadily increasing clock speed are long gone

Moore’s law provides instead increased parallelism

- More cores (later)
- Wider SIMD registers
- New instructions
- …

But software needs to keep up with the hardware

- Tools can help

You might have heard this before ☺

Will it vectorize effectively?

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

```
for (j=1; j<MAX; j++)  a[j]=a[j-n]+b[j];

for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];

for (int j=0; j<SIZE; j++)  {
    for (int i=0; i<SIZE; i++)   b[i] += a[i][j] * x[j];

for (int i=0; i<SIZE; i++)  b[i] += a[i] * x[index[i]];

for (j=1; j<MAX; j++)  sum = sum + a[j]*b[j]

for (int i=0; i<length; i++)   {
    float s = b[i]*b[i] - 4.f*a[i]*c[i];
        if ( s >= 0 ) x2[i] = (-b[i]+sqrt(s))/(2.*a[i]);
}
```

Will it vectorize effectively? Answers

1) Vectorizes if $n \leq 0$; doesn't vectorize if $n > 0$ and small; may vectorize if $n \geq$ 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) 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. gcc needs -ffast-math in addition.

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

Problems with Pointers

Hard for compiler to know whether arrays or pointers might be aliased (point to the same memory location)

- Aliases may hide dependencies that make vectorization unsafe
- Bigger problem for C than Fortran (which has TARGET attribute)

In simple cases, compiler may generate vectorized and unvectorized loop versions, and test for aliasing at runtime

Otherwise, compiler may need help:

- -fargument-noalias & similar switches
- __restrict__ or "restrict" keyword with -restrict or -std=c99 or by inlining
- #pragma ivdep asserts no potential dependencies
  - Compiler still checks for proven dependencies
- #pragma omp simd asserts no dependencies, period

```
void saxpy (float *x, float *y, float*restrict z, float *a, int n) {
#pragma ivdep
  for (int i=0; i<n; i++) z[i] = *a*x[i] + y[i];
}
```

Example of New Optimization Report

```
ifort -c -xavx -qopt-report-phase=loop,vec -qopt-report-file=stderr -qopt-report=3 func1.f90

LOOP BEGIN at func1.f90(9,25)
<Multiversioned v1>
  remark #25233: Loop multiversioned for stride tests on Assumed shape arrays
  remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 2
  remark #15451: unmasked unaligned unit stride stores: 1
…
  remark #15478: estimated potential speedup: 3.080
…
LOOP END

LOOP BEGIN at func1.f90(9,25)
<Multiversioned v2>
  remark #15300: LOOP WAS VECTORIZED
  remark #15460: masked strided loads: 2
  remark #15462: unmasked indexed (or gather) loads: 1
…
  remark #15478: estimated potential speedup: 2.330
…
LOOP END
…
```

```fortran
subroutine func( theta, sth )
  integer, parameter :: fp=8
  real(fp), parameter :: pi=acos(-1._fp)
  real(fp), parameter :: d180=180._fp
  real                :: theta(:), sth(:)
  integer             :: i

  do i=1,128
    sth(i) = sth(i) + (theta(i) * pi / d180)
  end do
end
```

Assumed shape array arguments may not be contiguous

Optimization Report Example

Convert to assumed size arguments (theta(*), sth(*)), or inline, or
use **CONTIGUOUS** keyword

Compiler knows arrays are contiguous, ⇒ only one loop version

```
ifort -c -xavx -qopt-report-phase=loop,vec -qopt-report-file=stderr -qopt-report=4 func2.f90
…
  remark #15305: vectorization support: vector length 4
  remark #15399: vectorization support: unroll factor set to 8
…
LOOP BEGIN at func2.f90(9,25)
  remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 2
  remark #15451: unmasked unaligned unit stride stores: 1
….
  remark #15478: estimated potential speedup: 3.080
…
  remark #15487: type converts: 3
  remark #25015: Estimate of max trip count of loop=4
LOOP END
```

```
subroutine func( theta, sth )
  integer, parameter :: fp=8
  real(fp), parameter :: pi=acos(-1._fp)
  real(fp), parameter :: d180=180._fp
  real, contiguous     :: theta(:), sth(:)
  integer              :: i

  do i=1,128
     sth(i) = sth(i) + (theta(i) * pi / d180)
  end do
end
```

Don’t take speedup estimates too seriously!!

- There’s a lot the compiler doesn’t know

Optimization Report Example

Keep all variables at the same precision

- Avoid unnecessary conversions
- Use full width of vector register

```
$ ifort -c -xavx -qopt-report-phase=loop,vec -qopt-report-file=stderr -qopt-report=4 func3.f90
…
  remark #15305: vectorization support: vector length 8
  remark #15399: vectorization support: unroll factor set to 8
…
LOOP BEGIN at func3.f90(9,25)
  remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 2
  remark #15451: unmasked unaligned unit stride stores: 1
…
  remark #15478: estimated potential speedup: 4.560
…
  remark #25015: Estimate of max trip count of loop=4
LOOP END
```

```
subroutine func( theta, sth )
  integer, parameter :: fp=4
  real(fp), parameter :: pi=acos(-1._fp)
  real(fp), parameter :: d180=180._fp
  real, contiguous     :: theta(:), sth(:)
  integer              :: i

  do i=1,128
    sth(i) = sth(i) + (theta(i) * pi / d180)
  end do
end
```

Don’t take speedup estimates too seriously!!

- There’s a lot the compiler doesn’t know

Optimization Report Example

Aligned loads and stores are more efficient than unaligned
Compiler doesn’t know alignment for dummy arguments

⇒ If you know the data are aligned, tell the compiler

```
$ ifort -c -xavx -qopt-report-phase=loop,vec -qopt-report-file=stderr -qopt-report=3 func4.f90
…
LOOP BEGIN at func4.f90(8,3)
  remark #15300: LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 2
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector loop cost summary ---
  remark #15476: scalar loop cost: 34
  remark #15477: vector loop cost: 39.000
  remark #15478: estimated potential speedup: 6.970
  remark #15488: --- end vector loop cost summary ---
  remark #25015: Estimate of max trip count of loop=4
LOOP END
```

```
subroutine func( theta, sth )
  integer, parameter :: fp=4
  real(fp), parameter :: pi=acos(-1._fp)
  real(fp), parameter :: d180=180._fp
  real, contiguous    :: theta(:), sth(:)
  integer             :: i
!dir$ assume_aligned theta:32,sth:32
  do i=1,128
     sth(i) = sth(i) + (theta(i) * pi / d180)
  end do
end
```

Don’t take speedup estimates too seriously!!

- There’s a lot the compiler doesn’t know

See webinar for more detail and explanation of optimization reports

From the Old Days, recap…

Requirements for Auto-Vectorization

Innermost loop of nest
Straight-line code

Avoid:

- Function/subroutine calls
- Loop carried data dependencies
- Non-contiguous data (indirect addressing; non-unit stride)
- Inconsistently aligned data

See   http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/

Still good advice, but no longer absolute requirements

Explicit Vector Programming:

Use #pragma omp simd or !$OMP SIMD   with -qopenmp-simd

```
void addit(double* a, double* b,
int m, int n, int x)
{
  for (int i = m; i < m+n; i++)   {
        a[i] = b[i] + a[i-x];
  }
}
```

loop was not vectorized:
existence of vector dependence.

```
void addit(double* a, double * b,
int m, int n, int x)
{
#pragma omp simd  // I know x<0
  for (int i = m; i < m+n; i++)   {
        a[i] = b[i] + a[i-x];
  }
}
```

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-pragma-simd/

Minimizes source code changes needed to enforce vectorization

SIMD-enabled Function

Compiler generates vector version of a scalar function that can be called from a vectorized loop:

```
#pragma omp declare simd (uniform(y,z,xp,yp,zp))
float func(float x, float y, float z, float xp, float yp, float zp)
{
float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp) + (z-zp)*(z-zp);
  denom = 1./sqrtf(denom);
  return denom;
}
....
#pragma omp simd  private(x) reduction(+:sumx)
for (i=1;i<nx;i++) {
   x = x0 + (float)i*h;
   sumx = sumx + func(x,y,z,xp,yp,zp);
 }
```

y, z, xp, yp and zp are constant,
x can be a vector

FUNCTION WAS VECTORIZED with…

These clauses are required for correctness, just like for OpenMP*

SIMD LOOP WAS VECTORIZED

Prefetching - automatic

Compiler prefetching is very important for the Intel® Xeon Phi™ coprocessor; not so important for Intel® Xeon processors

- Knights Landing is somewhere in between
- Can be very important for apps with many L2 cache misses

-qopt-prefetch=n (4 = most aggressive) to control globally

-qopt-prefetch=0 or -qno-opt-prefetch to disable

-qopt-prefetch-distance =<n1> [,<n2>] to tune how far ahead to prefetch

- n1 is for prefetch to L2; n2 is for (second level) prefetch to L1 from L2
- Units are loop iterations

Use the compiler reporting options to see detailed diagnostics of prefetching per loop

-qopt-report-phase=loop -qopt-report=3 e.g.

```
Total number of lines prefetched=2
Using noloc distance 8 for prefetching unconditional memory reference    [ fun.F90(42,27) ]
Using second-level distance 2 for prefetching unconditional memory reference   [ fun.F90(42,27) ]
```

Prefetching - manual

Use intrinsics

```
_mm_prefetch((char *) &a[i], hint);
```

See xmmintrin.h for possible hints (for L1, L2, non-temporal, …)

`MM_PREFETCH(A, hint)` for Fortran

- But you have to figure out and code how far ahead to prefetch
- Also gather/scatter prefetch intrinsics, see zmmintrin.h and compiler user guide, e.g. _mm512_prefetch_i32gather_ps

Use a pragma / directive (easier):

```
#pragma prefetch  a   [:hint[:distance]]
!DIR$ PREFETCH A, B, …
```

- You specify what to prefetch, but can choose to let compiler figure out how far ahead to do it.

`!DIR$ PREFETCH  *` to prefetch all arrays in loop

Hardware prefetching is also enabled by default

- On Intel® Xeon, beware of too much software prefetching

Auto-vectorization and report

Traditional programming with arrays:

```
#include <math.h>

double no_stl (double * x, int n) {
	double sum{ 0.0 };
	for (int i=0; i<n; i++)
		sum += sqrt(sin(x[i])*sin(x[i]) + cos(x[i])*cos(x[i]));
	return sum;
}
```

```
icpc -c -qopt-report-phase=vec -qopt-report=3 no_stl.cpp
...
LOOP BEGIN at no_stl.cpp(5,2)
   remark #15300: LOOP WAS VECTORIZED
...
 remark #15481: heavy-overhead vector operations: 1
 remark #15482: vectorized math library calls: 1
...
grep cos no_stl.s
   call      __svml_sincos2
```

Optimization report goes to no_stl.optrpt
(or use -qopt-report-file=stderr to see directly)

Call to sincos is "heavy overhead" (takes a long time), so a good candidate for speed-up by being vectorized

STL vectors - do they vectorize?

STL containers are a popular and portable programming method.

- The loop is written using iterators over the vector class
- No need to pass the vector size explicitly

```
#include <vector>
#include <math.h>

double stl_vec(std::vector<double>& x) {
        double sum{ 0.0 };

        for (std::vector<double>::iterator ix=x.begin(); ix!=x.end(); ix++)
              sum += sqrt(sin(*ix)*sin(*ix) + cos(*ix)*cos(*ix));
        return sum;
}
```

```
icpc -c -qopt-report-file=stderr -qopt-report-phase=vec -qopt-report=2  stl_vec.cpp
…
LOOP BEGIN at stl_vec.cpp(6,53)
  remark #15300: LOOP WAS VECTORIZED
```

Performance is closely similar to that with arrays

STL vectors - using C++11

C++11 introduced two convenient new features:

- The **`auto`** keyword
- Range-based for loops

```
#include <vector>
#include <math.h>

double stl_vec_11 (std::vector<double>& x) {
      double sum{ 0.0 };

      for (auto ix : x)
            sum += sqrt(sin(ix)*sin(ix) + cos(ix)*cos(ix));
      return sum;
}
```

```
icpc -c -std=c++11 -qopt-report-phase=vec -qopt-report=3 stl_vec_11.cpp
…
LOOP BEGIN at stl_vec.cpp(6,53)
  remark #15300: LOOP WAS VECTORIZED
```

Both the type and the values of the loop parameters are taken from the class

STL vectors - alignment

Aligned data accesses are more efficient.

So can we align our data? (to 32 byte boundaries for Intel® AVX)

```
    typedef vector<double>  dvec;
     __attribute__((align(32)))  My_dvec x(SIZE,0.6);
```

or

```
    dvec *xx = new(_mm_malloc(sizeof(dvec(SIZE)), 32))  dvec(SIZE, 0.6);
```

- No! This aligns the vector class, not the data member (see test program)
  - Compile with -DALIGN_CLASS (optionally with -DDYNAMIC) to test
- boost 1.56 or later provides an aligned allocator that can be used to align the data member:

```
#include <boost/align/aligned_allocator.hpp>
typedef std::vector<double, boost::alignment::aligned_allocator<double, 32> > dvec;
```

  - This aligns data member but not the class itself
  - Compile with -DBOOST_ALIGN (optionally with -DDYNAMIC) to test
  - Add #pragma vector aligned to tell the compiler

An executable example

See the source file sumsq.cpp

Start with

```
typedef vector<double> My_dvec;
        My_dvec x(SIZE,0.6);
        My_dvec y(SIZE,0.8);
Or
        My_dvec *xx = new My_dvec(SIZE, 0.6);
        My_dvec *yy = new My_dvec(SIZE, 0.8);
```

Build and run with

```
icpc -O2 -std=c++11 -I$BOOST/boost_1_56   sumsq.cpp  timer.cpp;  ./a.out
```

Prints out alignment of the class and the first element of the vector
Sums the squares of the elements of the vectors
Prints the execution time

Align the vector data member, not the class itself

Want 32 byte alignment for Intel® AVX2. Compile and run example with:

icpc -O2 -xcore-avx2 -std=c++11 -I$BOOST/boost_1_56 sumsq.cpp timer.cpp; ./a.out

```
    alignment of     x   y       preprocessor macro        run time

vector alignment:   16   0
class  alignment:    0  24

vector alignment:   16   0       -DALIGN_CLASS               0.60 s
class  alignment:    0   0

vector alignment:    0   0       -DBOOST_ALIGN               0.55 s
class  alignment:   16   8

vector alignment:   16   0       -DDYNAMIC                   0.60 s
class  alignment:   16   0

vector alignment:    0   0       -DDYNAMIC -DBOOST_ALIGN  0.55 s
class  alignment:   16  16
```

similar behavior for static or dynamic allocation. “0” means 32 byte aligned.

STL vectors - performance

Look at more detail:

```
icpc -c  -std=c++11 -xcore-avx2 -qopt-report-file=stderr -qopt-report-phase=loop,vec
                             -qopt-report3 -qopt-report-routine=main  sumsq.cpp
...
  LOOP BEGIN at sumsq.cpp(95,31) inlined into sumsq.cpp(73,43)
  <Peeled>
  LOOP END

  LOOP BEGIN at sumsq.cpp(95,31) inlined into sumsq.cpp(73,43)
    remark #15300: LOOP WAS VECTORIZED
    remark #15450: unmasked unaligned unit stride loads:
  LOOP END
...
```

Peel loop: compiler can align accesses to one array dynamically at run-time, but not to two.

Vectorized loop kernel

Add -DBOOST_ALIGN and this becomes:

```
  LOOP BEGIN at sumsq.cpp(95,31) inlined into sumsq.cpp(73,43)
    remark #15300: LOOP WAS VECTORIZED
    remark #15450: unmasked aligned unit stride loads: 2
```

sumsq () inlined into main

"Peel" loop is only generated when compiler doesn't know alignment
Accesses to data known to be aligned are (modestly) faster than to unaligned data
For more about peel and remainder loops, see https://software.intel.com/videos/getting-the-most-out-of-the-intel-compiler-with-new-optimization-reports

STL vectors - performance

Optimization Options	Speed-up	What's going on
-O0	0.014	no inlining
-O1	0.66	some inlining restrictions
-O2 -no-vec	1.0	no vectorization
-O2	1.9	vectorization
-O2 -xavx	3.2	wider vectors
-O2 -xcore-avx2	4.2	Fused multiply-add
-O2 -xcore-avx2 -DALIGN_CLASS	4.2	Class aligned
-O2 -xcore-avx2 -DBOOST_ALIGN	4.5	Data member aligned

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. Boost* version 1.56 and the Intel® C++ Compiler version 16.0 beta were used.

Computations on Arrays

```
#define SIZE 10000
#define NTIMES 10000
void test(std::array<float,SIZE> &vi, std::array<float,SIZE> &va)
{
   for(int i = 0; i < SIZE; i++)
      vi[i] = 2.f*sin(va[i]);
}


              int main()
{
   std::array<float,SIZE>  vi;
   std::array<float,SIZE>  va;
   float mysum;
   int i,j, size1, ntimes;
   double execTime = 0.0;
   double startTime, endTime;
   size1=SIZE;
   ntimes=NTIMES;
   for (int i=0; i<SIZE; i++) {
               vi[i] = (float) i;
               va[i] = (float)(i+10);
              }
   startTime = clock_it();
   for (j=0; j<ntimes; j++) {
     test(vi, va);
   }
   endTime = clock_it();
..
..
}
```

•Every scientific computing application deals with really huge arrays of either floats or doubles.

•Code snippet computes the sine value of every element in "va" array multiplied by 2 and stores it in corresponding "vi" array location.

```
$ icpc driver1.cc test1.cc -xavx -std=c++11 -o test2
$ ./test2
sum= -4.012442   -1.088042   -0.227893
time = 271 ms
```

Converting the program to use Valarray

```
#define SIZE 10000
#define NTIMES 10000
void test(std::valarray<float> &vi, std::valarray<float> &va)
{
 vi = 2.f*sin(va);
}

            int main()
{
   std::valarray<float>  vi(SIZE);
   std::valarray<float>  va(SIZE);
   float mysum;
   int i,j, size1, ntimes;
   double execTime = 0.0;
   double startTime, endTime;
   size1=SIZE;
   ntimes=NTIMES;
   for (int i=0; i<SIZE; i++) {
              vi[i] = (float) i;
              va[i] = (float)(i+10);
            }
   startTime = clock_it();
   for (j=0; j<ntimes; j++) {
     test(vi, va);
   }
   endTime = clock_it();
..
..
}
```

- Valarray is part of standard C++ library provided by both GNU C++ and Microsoft.
- Valarray provides operator overloaded functions for basic operators and math functions.
- Unlike the previous case, the function test() doesn't need a "for loop" to iterate through the individual elements.
- Introduced mainly to help with huge array based computations in High Performance Computing space.

```
$ icpc driver.cc test.cc -xavx -o test1
$ ./test1
sum= -4.012444   -1.088042   -0.227893
```

time = 136 ms

Speedup w.r.t array version ~ 2x

Use Intel® Integrated Performance Primitives version of Valarray

- Intel® Integrated Performance Primitives provides a more optimal optimization.
- valarray header is provided by Intel® C++ Compiler : <install_dir>/perf_headers
- Intel® C++ Compiler by default uses valarray header provided by GNU compiler on Linux* and Microsoft Compiler on Windows*.
- To use the Intel optimized header (ippvalarray), use the compiler option -use-intel-optimized-headers

```
$ icpc driver.cc test.cc -xavx -o test3 -std=c++11 -use-intel-optimized-headers
$ ./test3
sum= -4.012444   -1.088042   -0.227893
time = 94 ms
```

Speedup w.r.t previous valarray version = 1.46x
Speedup w.r.t array version = 2.91x

System Specifications:

Processor: Intel(R) Core(TM) i5-4670T CPU @ 2.30GHz
RAM: 8GB
Operating System: Ubuntu* 12.04 (Linux* kernel - 3.8.0-29-generic)
GCC Compatibility mode : 4.6.3
Intel® Compiler Version: 16.0

Article URL: https://software.intel.com/en-us/articles/using-improved-stdvalarray-with-intelr-c-compiler