INTEL® COMPILER OPTIMIZATION AND BUILDING FOR KNL

Software Solutions Group
Intel® Corporation

Agenda

Optimization:

- **Brief overview**
- Advanced vectorization

Building for KNL:

- New optimization opportunities with Intel® AVX-512
- Allocating high bandwidth memory

Basic Optimizations with icc or ifort -O...

-O0 no optimization; sets -g for debugging

-O1 scalar optimizations

- Excludes optimizations tending to increase code size

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

- includes **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

-qopt-report[=0-5] reports what optimizations were performed

gfortran, gcc most optimizations off by default

-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

InterProcedural Optimization (IPO)

icc -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)
- Optimized paths for Intel® AVX2 and Intel® AVX-512 (detected at run-time)

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

Options to control precision and "short cuts" for vectorized math library:

- -fimf-precision = < high | **medium** | low >
- -fimf-domain-exclusion = < mask >
  - Library need not check for special cases (∞, nan, singularities)

Agenda

Optimization:

- Brief overview
- **Advanced vectorization**

Building for KNL:

- New optimization opportunities with Intel® AVX-512
- Allocating high bandwidth memory

Some General Advice for Auto-Vectorization

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

Disambiguate function arguments for C/C++

- E.g. By using -fargument-noalias or “restrict”

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
  - Compiler must also worry about pointer aliasing

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 have unit stride, tell the compiler:

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

May sometimes need **contiguous** also in caller to avoid temporary array copy

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

Vectorization is so important for performance

➔ 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

  #pragma omp simd or !$OMP SIMD etc.
  - Programmer is responsible for correctness (just like OpenMP threading)
    - E.g. PRIVATE and REDUCTION clauses
  - Overrides all dependencies and cost-benefit analysis
- Now incorporated in OpenMP 4.0 ⇒ portable
  - -qopenmp or -qopenmp-simd to enable

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
- REDUCTION
- COLLAPSE

} like OpenMP for threading (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 “for” or “DO” loop
  - SIMD-enabled function may be called with array section argument
    - For Fortran, also needs “ELEMENTAL” keyword

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

19

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, ...

20

Use PROCESSOR clause to get full benefit on KNL (with 17.0 compiler)

#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.

23

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 optimization reports
  - With explicit vector programming
  - Use Intel® Advisor and/or Intel® VTune™ Amplifier to find the best places (hotspots) to focus your efforts
- No need to re-optimize vectorizable code for new processors
  - Typically a simple recompilation

Agenda

Optimization:

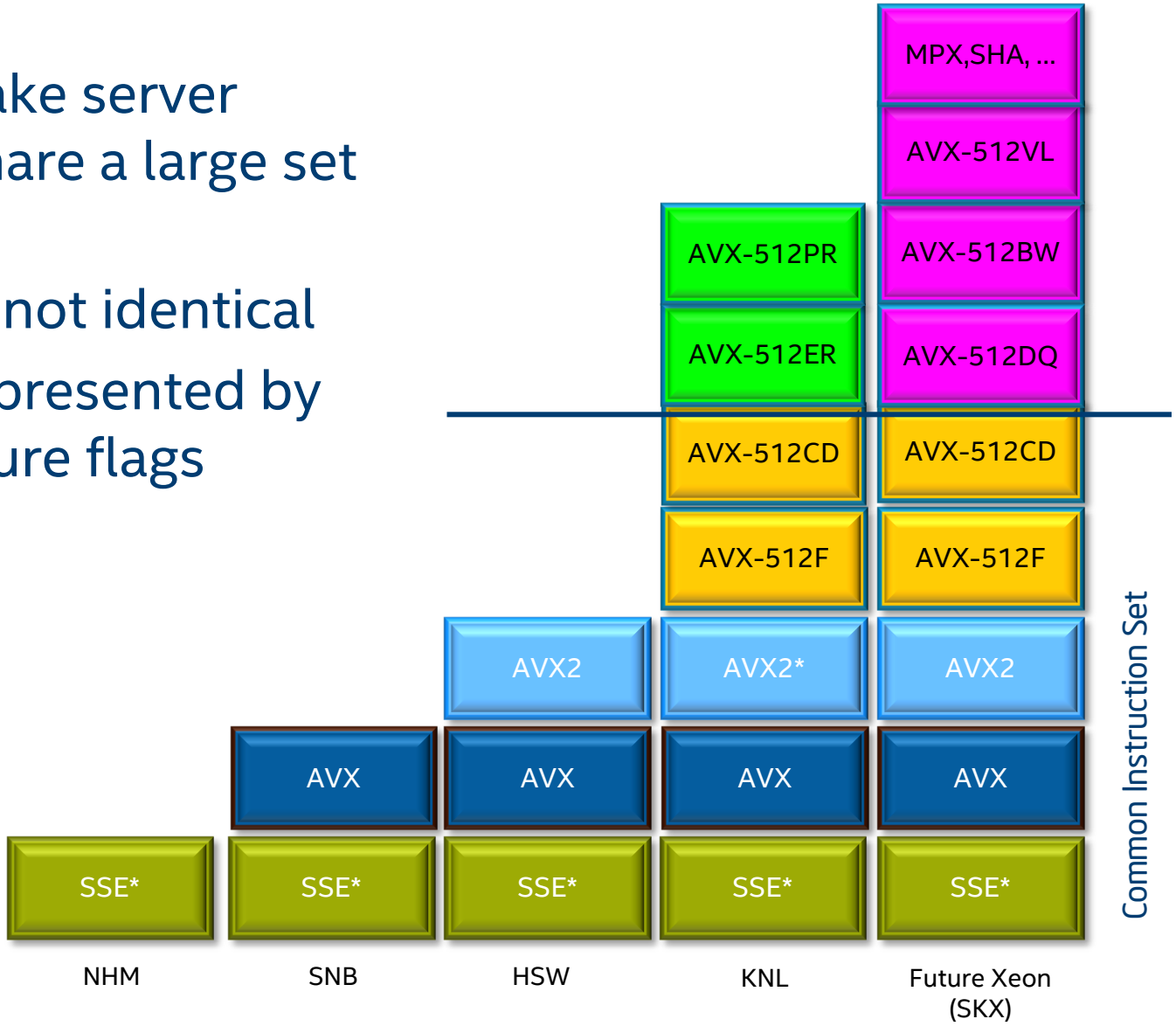
- Brief overview
- Advanced vectorization

Building for KNL:

- **New optimization opportunities with Intel® AVX-512**
- Allocating high bandwidth memory

Reminder: AVX-512 – KNL and SKX

- KNL and SkyLake server 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. Not 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

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=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.

- I saw about 8x for my single precision test case

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

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

- ~9x in my little example
- 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
```

Bottom Line for FP consistency

To get consistent results between KNL and Intel Xeon processors, use

-fp-model precise -fimf-arch-consistency=true -no-fma

(you could try omitting -no-fma for Xeon processors that support FMA, but FMA's could still possibly lead to differences)

In the 17.0 compiler, this can be done with a single switch:

- -fp-model consistent

To get consistent results that are as close as possible between KNC and Intel® Xeon® processors or KNL, try

-fp-model precise -no-fma on both.

FMAs

Most common cause of floating-point differences between Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors or KNL

- Not disabled by -fp-model precise
- Can disable for testing with -no-fma
- Or by function-wide pragma or directive:

```
#pragma float_control(fma,off)
!dir$ nofma
```

  - With some impact on performance
- -fp-model strict disables FMAs, amongst other things
  - But on KNC, results in non-vectorizable x87 code
- The fma() intrinsic in C should always give a result with a single rounding, even on processors with no FMA instruction

FMAs

Can cause issues even when both platforms support them (e.g. Haswell and KNL)

- Optimizer may not generate them in the same places
  - No language rules
- FMAs may break the symmetry of an expression:

```
c = a;    d = -b;
result = a*b + c*d;       ( = 0  if no FMAs )
```

If FMAs are supported, the compiler may convert to either

result = fma(c, d, (a*b)) or result = fma(a, b, (c*d))

Because of the different roundings, these may give results that are non-zero and/or different from each other.

Agenda

Optimization:

- Brief overview
- Advanced vectorization

Building for KNL:

- New optimization opportunities with Intel® AVX-512
- **Allocating high bandwidth memory**

High Bandwidth 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)

- 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 additions in a 17.0 compiler:

- Attaching new FASTMEM directive to ALLOCATE statement
  - Instead of ALLOCATABLE declaration

C++: can pass hbw_malloc() etc.

standard allocator replacement for e.g. STL like

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

Available already, working on documentation

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 previous 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

How much HBM is left?

```
#include <memkind.h>
int hbw_get_size(int partition, size_t * total, size_t * free) {       // partition=1  for flat 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: (use Fortran 2003 C-interoperability features)

```
 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

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

48

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

Additional Resources (KNL & General)

https://software.intel.com/articles/xeon-phi-software

https://software.intel.com/articles/intel-xeon-phi-coprocessor-code-named-knights-landing-application-readiness

https://software.intel.com/sites/default/files/managed/4c/1c/parallel_mag_issue20.pdf

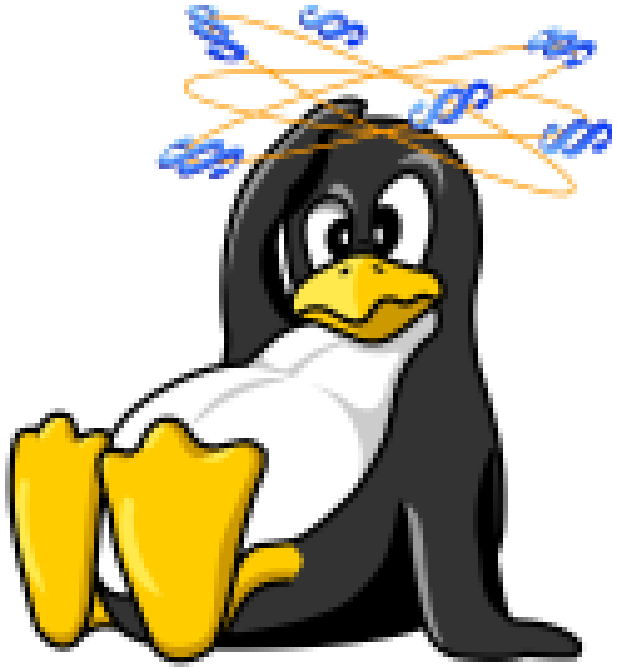
https://software.intel.com/articles/intel-software-development-emulator

https://github.com/memkind

https://software.intel.com/articles/consistency-of-floating-point-results-using-the-intel-compiler

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

Questions?



Legal Disclaimer & Optimization Notice

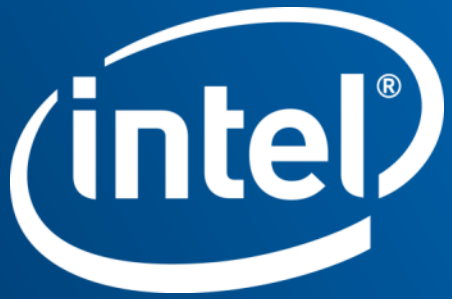
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING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

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

Optimization Notice

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

Notice revision #20110804


intel