



BUILDING YOUR APPLICATION FOR THE INTEL® XEON PHI™ X200 PROCESSOR

Formerly Code named Knights Landing

Agenda

Intel's Optimizing Compiler: Very Brief Overview/Recap

(examples are for Linux* but behavior is the same for Windows*)

Intel® Xeon Phi™ x200 Processor Overview

(formerly code named Knights Landing, sometimes abbreviated to KNL to save space)

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

High Bandwidth Memory

Consistency of Floating-Point Results

Intel® Compiler: Brief Recap

Supports standards

- Fortran77, Fortran90, Fortran95, Fortran2003, much Fortran 2008
- Up to C99, C++11; C++ 14; Minimal C11 and C++17 (so far)
 - `-std=c99` `-std=c++11` `-std=c++14` `-std=c++17`

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

- But not binary compatible with gfortran

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

OpenMP* 4.0 support, much 4.5, no user-defined reductions

Optimized math libraries (including for KNL)

Many advanced optimizations

- With detailed, structured optimization reports

Drivers: `icc`, `icpc`, `ifort`

- To set the environment: `source compilervars.sh intel64`

Basic Optimizations with icc/ifort -O...

- O0 no optimization; sets -g for debugging
- O1 scalar optimizations
 - Excludes optimizations tending to increase code size
- O2 **default** (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
- qopt-report [=0-5]
 - Generates compiler optimization reports in files *.optrpt

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 C++ 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; faster than GNU libm) and libsvml (vector)
- 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

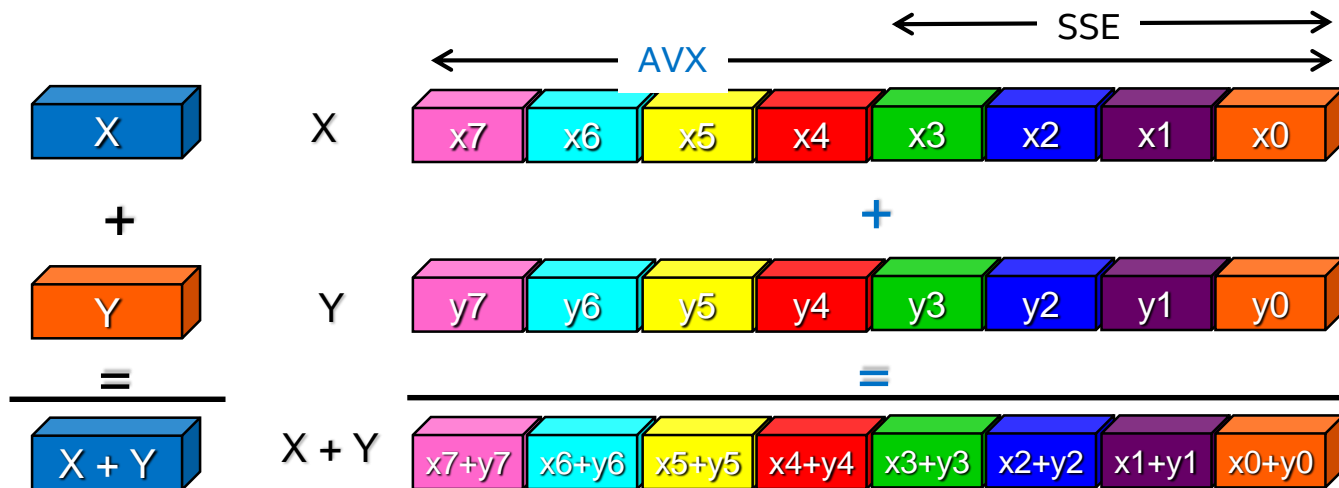
Options to control precision and “short cuts” for vectorized math library:

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

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. `vaddss`, `vaddsd`
- Vector (SIMD) mode
 - one instruction can produce multiple results
 - E.g. `vaddps`, `vaddpd`



Guidelines for Writing Vectorizable Code

Prefer simple “for” or “DO” loops

Write straight line code. Try to avoid:

- function calls (unless inlined or SIMD-enabled functions)
- 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

- 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.
- Disambiguate function arguments, e.g. -fargument-noalias

Use efficient memory accesses

- Favor inner loops with unit stride
- Minimize indirect addressing `a[i] = b[ind[i]]`
- Align your data consistently where possible (to 16, 32 or 64 byte boundaries)

Explicit SIMD (Vector) Programming

Modeled on OpenMP* for threading (explicit parallel programming)

```
#pragma omp simd <clauses>      (for loops)
#pragma omp declare simd <clauses>  (for functions)
```

Enables reliable vectorization of complex loops that compiler can't auto-vectorize

- E.g. outer loops

Directives are commands to the compiler, not hints

- Programmer is responsible for correctness (like OpenMP threading)
 - E.g. PRIVATE and REDUCTION clauses
- Overrides all dependency and cost-benefit analysis

Incorporated in OpenMP 4.0 \Rightarrow portable

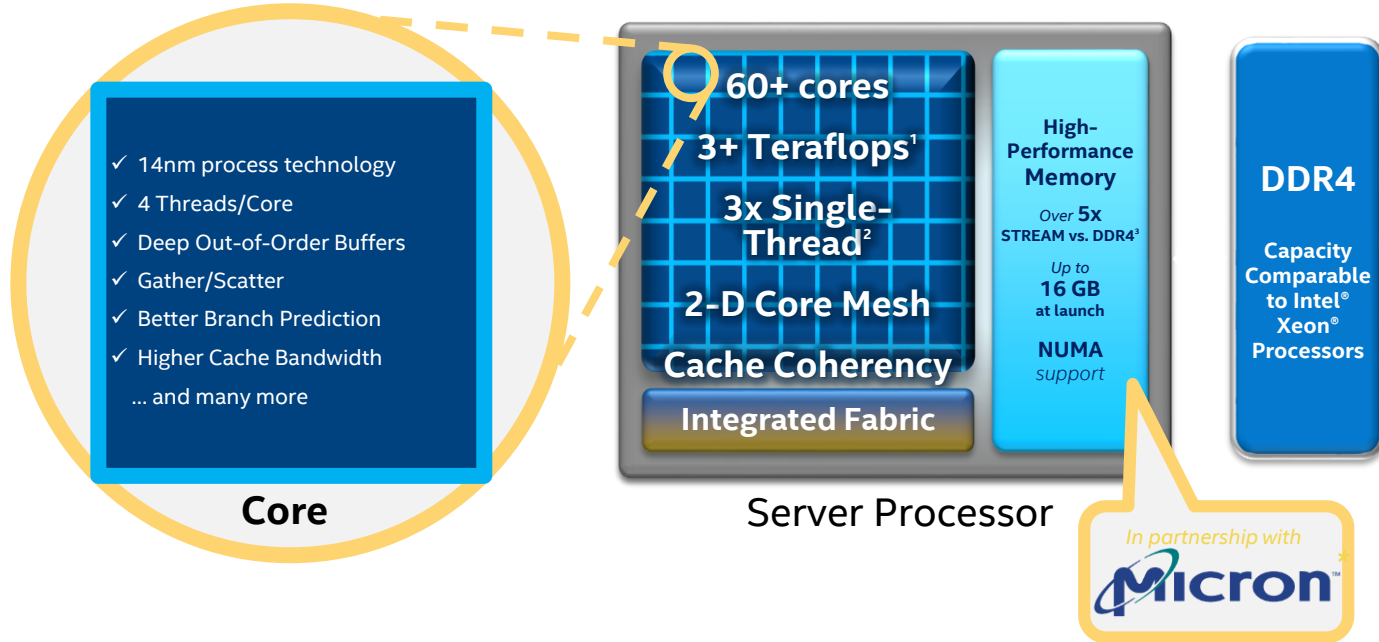
- `-qopenmp` or `-qopenmp-simd` to enable

```
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];
}
```

Knights Landing: Next-Generation Intel® Xeon Phi™

Architectural Enhancements = ManyX Performance

*Based on
Intel® Atom™ core (based
on Silvermont
microarchitecture) with
Enhancements for HPC*



Optimization Notice

Copyright © 2017, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.



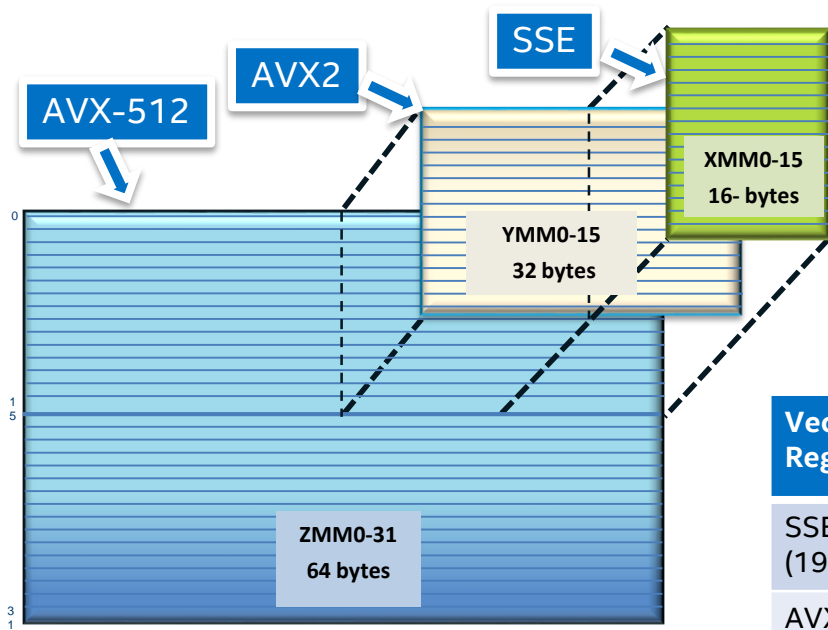
Processor Overview

- Self- or leveraged-boot
 - Self-boot may be easier to use (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

Intel® 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 |

Optimization Notice

Copyright © 2017, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

The Intel® AVX-512 Subsets [1]

AVX-512 F

AVX-512 F: 512-bit **F**oundation 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 (**C**onflict **D**etection instructions)

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

AVX-512ER

AVX-512 extensions for exponential and prefetch operations

AVX-512PR

- ❑ fast (28 bit) instructions for **e**xponential and **r**eciprocal (as well as RSQRT)
- ❑ New **p**refetch 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

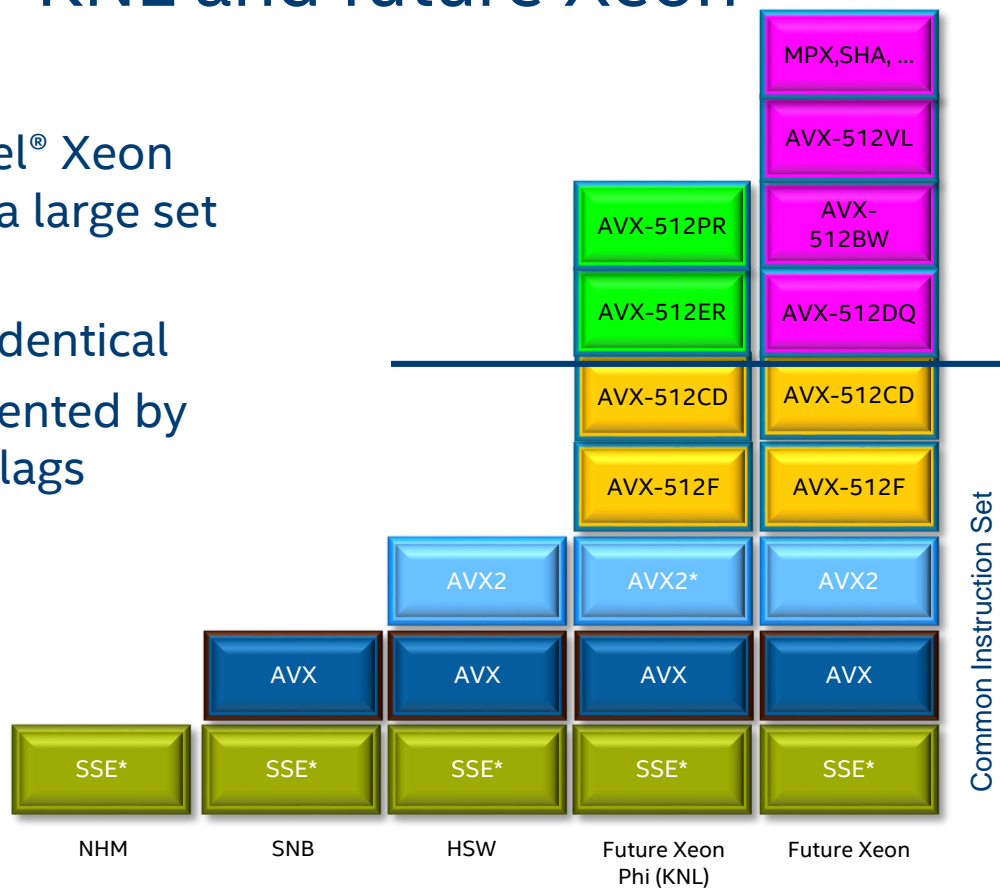
- ❑ needed for future memory technologies

XSAVE(S,C)

Save and restore extended processor state

Intel® 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 (Intel® AVX-512CD)

Compress/Expand Loops VECTORIZABLE with Intel® AVX-512

```
nb = 0
do ia=1, na           ! line 23
  if(a(ia) > 0.) then
    nb = nb + 1       ! dependency
    b(nb) = a(ia)     ! compress
  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 via j and k
```

With Intel® AVX2, does not auto-vectorize

- and vectorizing with an OpenMP* SIMD directive would be unsafe

```
ifort -c -xcore-avx2 -qopt-report-file=stderr -qopt-report=3 -qopt-report-phase=vec compress.f90
```

```
...
LOOP BEGIN at compress.f90(23,3)
  remark #15344: loop was not vectorized: vector dependence prevents vectorization. ...
  remark #15346: vector dependence: assumed ANTI dependence between nb (25:7) and nb (25:7)
LOOP END
```

- C code behaves the same

Compress Loop

Compile for Intel® AVX-512:

```
ifort -c -qopt-report-file=stderr -qopt-report=3 -qopt-report-phase=vec -xmic-avx512 compress.f90
```

```
...  
LOOP BEGIN at compress.f90(23,3)  
  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
```

- Compile with `-S` to see new instructions in assembly code:

```
grep vcompress compress.s
```

| | |
|--|-------------------------------|
| <code>vcompressps %zmm4, -4(%rsi,%rdx,4){%k1}</code> | <code>#14.7 c7 stall 1</code> |
| <code>vcompressps %zmm1, -4(%rsi,%r12,4){%k1}</code> | <code>#14.7 c5</code> |
| <code>vcompressps %zmm1, -4(%rsi,%r12,4){%k1}</code> | <code>#14.7 c5</code> |
| <code>vcompressps %zmm4, -4(%rsi,%rdi,4){%k1}</code> | <code>#14.7 c7 stall 1</code> |

Compress Loop: speed-up

- Run for 1,000,000 elements, repeated 1000 times:
 - `ifort -xcore-avx2 -qopt-report=3 driver.F90 compress.f90; ./a.out`
 - 13 secs
 - `ifort -xmhc-avx512 -qopt-report=3 driver.F90 compress.f90; ./a.out`
 - 0.8 secs
 - Similar for C version
- Speed-up depends on compression factor
 - Less for high compression

Intel® Xeon Phi™ 7250 processor
1.4 GHz 68 cores
Red Hat* EL 7.2
Intel® Compiler 17.0


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

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

INTEL® AVX-512 Conflict Detection Instructions

The **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

VPCONFLICT instr.

VPCONFLICT{D,Q} zmm2/mem, zmm1{k1}

VPTESTNM{D,Q} zmm2, zmm3/mem, zmm2, k2{k1}

VPBROADCASTM{W2D,B2Q} k2, zmm1

VPLZCNT{D,Q} zmm2/mem, zmm1 {k1}

```
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
enddo
```

```
for (i=0; i<n; i++) {
  y    = sinf(x[i]*twopi);
  ih   = floor((y-bot)*invbinw);
  ih   = ih > 0    ? ih : 0;
  ih   = ih < nbin ? ih : nbin-1;
  h[ih] = h[ih] + 1;
}
```

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...
```

```
  remark #15346: vector dependence: assumed FLOW dependence between line 15 and line 15
```

```
LOOP END
```

Histogramming with Intel® AVX-512 CD

Compile for Intel® Xeon Phi™ processor x200 family:

```
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 %zmm20, %zmm5, %zmm3
vpconflictd %zmm3, %zmm1
#          work on simd lanes without conflicts
vpgatherdd (%r13,%zmm3,4), %zmm6{%k1} # load h
vptestmd .L_2ilOfloatpacket.5(%rip), %zmm1, %k0
vpadd %zmm21, %zmm6, %zmm2          #increment h
...
vpbroadcastmw2d %k1, %zmm4
vplzcntd %zmm1, %zmm4
vptestmd %zmm1, %zmm5, %k0
```

```
..B1.18          # loop over simd lanes with conflicts
kmovw %r10d, %k1
vpbroadcastmw2d %k1, %zmm4
vpermd %zmm2, %zmm0, %zmm2{%k1}
vpadd %zmm21, %zmm2, %zmm2{%k1} #increment histo
vptestmd %zmm1, %zmm4, %k0{%k1}
kmovw %k0, %r10d
testl %r10d, %r10d
jne ..B1.18
...
vpscatterdd %zmm2, (%r13,%zmm3,4){%k1} # final store
```

Histogramming with Intel® AVX-512: speed-up

Run time for Intel® AVX2 (non-vectorized) : 59 secs

Intel® AVX-512 (vectorized) : 6.6 secs

Intel® Xeon Phi™ 7250 processor, 1.4 GHz
Red Hat* EL 7.2
Intel® Compiler 17.0
Performance depends on many factors,
see slide 26

Speed-up 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
- Similar behavior for C and Fortran versions

Other problems map to this

- E.g. energy deposition in cells in particle transport Monte Carlo simulation

Gather TO SHUFFLE (“G2S”) Optimization

or “Adjacent Gather Optimization”

```
for (j=0; j<n; j++) {  
    y[j] = x[j][1] + x[j][2] + x[j][3] + x[j][4] ...  
}
```

- 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 AVX2: gather of x[1][1], x[2][1], x[3][1], x[4][1],...
 - With AVX-512: SIMD loads of x[1][1], x[1][2], x[1][3], x[1][4] etc., followed by permutes to get back to x[1][1], x[2][1], x[3][1], x[4][1] etc.
 - Message in optimization report:
remark #34029: adjacent sparse (indexed) loads optimized for speed
- Large arrays of short vectors or structs are very common

G2S Example - ARRAY OF STRUCTURES

```
float sumsq( struct Point *ptvec, int n) {  
    float  t_sum = 0;  
    int i;  
    // #pragma omp simd reduction(+:t_sum)  
    // #pragma vector nog2s  
    for (i = 0; i < n; i++) {        // loop over points  
        t_sum += ptvec[i].x * ptvec[i].x;  
        t_sum += ptvec[i].y * ptvec[i].y;  
        t_sum += ptvec[i].z * ptvec[i].z;  
#ifdef VEC4  
        t_sum += ptvec[i].t * ptvec[i].t;  
#endif  
    }  
    return t_sum;  
}
```

```
struct Point {  
    float x;  
    float y;  
    float z;  
#ifdef VEC4  
    float t;  
#endif  
};
```

Calculate sum of squares
of components of a vector

Driver loops 100000 times over
an array of 10000 points

G2S EXAMPLE

From the version 15 compiler optimization report: (15.0.7.235)

```
$ icc -xmic-avx512 -qopt-report=4 test_g2s.c sumsq.c
```

LOOP BEGIN at sumsq.c(9,9)

remark #15415: vectorization support: gather was generated for the variable ptvec: strided by 3 [sumsq.c(10,22)]

remark #15415: vectorization support: gather was generated for the variable ptvec: strided by 3 [sumsq.c(10,35)]

...

remark #15300: LOOP WAS VECTORIZED

remark #15460: masked strided loads: 6

LOOP END

Problem: pt[0].x, pt[1].x and pt[2].x are not adjacent in memory

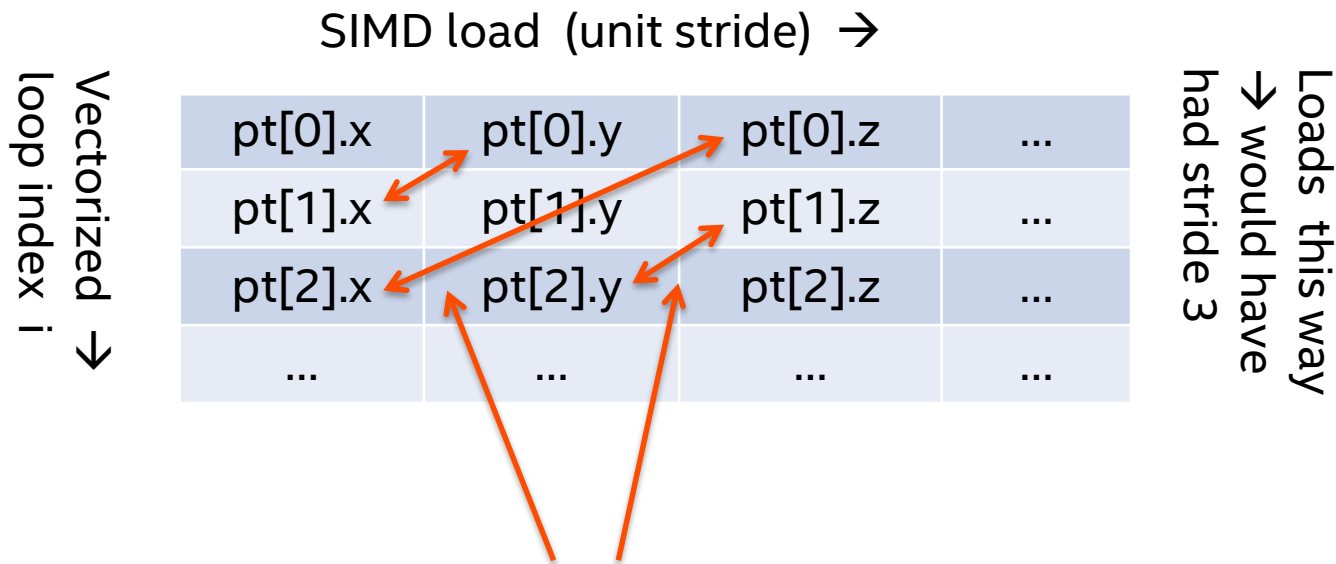
- \Rightarrow gather instructions are generated (slow)

pt[0].x, pt[0].y and pt[0].z are adjacent in memory, but we're not vectorizing in this dimension

- Much better to vectorize loop over many points than loop over 3 components

Run time 1.5 secs

Can we do better?



Transpose using Intel® AVX-512 permute instructions, vperm...
Enables unit stride SIMD loads as well as SIMD arithmetic
Supported in version 16 and 17 compilers

G2S Example:

From the version 17 compiler optimization report:

Compiler options: -xmic-avx512 -qopt-report=4

LOOP BEGIN at sumsq.c(9,9)

remark #15415: vectorization support: non-unit strided load was generated for the variable <ptvec->x[i], stride is 3

...

remark #15305: vectorization support: vector length 16

remark #15300: LOOP WAS VECTORIZED

remark #15452: unmasked strided loads: 6

...

Report from: Code generation optimizations [cg]

*sumsq.c(10,22):remark #34030: **adjacent sparse (strided) loads optimized for speed.***

Details: stride { 12 }, types { F32-V512, F32-V512, F32-V512 }, number of elements { 16 }, select mask { 0x000000007 }.

...

Run time 0.7 seconds

- Disable G2S with #pragma nog2s, to confirm it is responsible
- Run time reverts to 1.5 secs

Intel® Xeon Phi™ 7250 processor, 1.4 GHz
Red Hat* EL 7.2
Intel® Compiler 17.0
Performance depends on many factors,
see slide 26

A Bigger Struct

Suppose our points have 4 dimensions, not 3 (compile with -DVEC4)

- By default, compiler constructs and vectorizes an inner loop over components

LOOP BEGIN at sumsq.c(9,9)

remark #15542: loop was not vectorized: inner loop was already vectorized

...

LOOP BEGIN at sumsq.c(14,13)

...

remark #15305: vectorization support: vector length 4

remark #15427: loop was completely unrolled

remark #15301: MATERIALIZED LOOP WAS VECTORIZED

Run time 1.9 secs

- Inner loop trip count is short; better to vectorize outer loop

A Bigger Struct: outer loop vectorization

To enforce vectorization of outer loop, use:

```
#pragma omp simd reduction(+:t_sum)
```

```
icc -xmic-avx512 -qopenmp-simd -qopt-report=4 -DVEC4 test_g2s.c sumsq.c
```

LOOP BEGIN at sumsq.c(9,9)

remark #15415: vectorization support: non-unit strided load was generated for the variable <ptvec->x[i]>, stride is 4

remark #15305: vectorization support: vector length 16

remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

remark #15452: unmasked strided loads: 8

...

Report from: Code generation optimizations [cg]

sumsq.c(10,22):remark #34030: adjacent sparse (strided) loads optimized for speed. Details: stride { 16 }, types { F32-V512, F32-V512, F32-V512, F32-V512 }, number of elements { 16 }, select mask { 0x00000000F }.

We have outer loop vectorization back, followed by the G2S optimization

- Run time 0.8 secs

FROM INTEL® SSE OR INTEL® AVX TO INTEL® AVX-512:

Setting Expectations

Doubling or quadrupling the vector length to 512 bits can boost application performance, but typically by less than 2x or 4x

- Applications have scalar sections, so are subject to Amdahl's Law
- Some applications are limited by access to data
 - If throughput bound, high bandwidth memory may help
 - If latency bound, prefetching may help
- Loops may need larger trip counts to get full benefit

But gains from newly vectorized loops can be large (new instructions!)

- SIMD directives not yet applicable to new vector loop types
- Working on new clauses...

Application hotspots may change

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);` C
 - `MM_PREFETCH(A, hint)` Fortran

Gather Prefetch Example

```
void foo(int n, int* A, int *B, int *C) {  
    // pragma_prefetch var:hint:distance  
    #pragma prefetch A:1:3      // prefetch to L2 cache  3 iterations ahead  
    #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 lfatches 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
```


KNL HIGH BANDWIDTH MEMORY

Adapting software 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,`

- Flat or hybrid mode only
- More Fortran data types may be supported eventually
 - Global, local, stack or heap;
 - Currently just allocatable arrays (16.0) and pointers (17.0)
 - OpenMP private copies: preview in 17.0 update 1
 - Must remember to link with libmemkind !

Possible addition in a future 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() returns 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_WARN =T/F default False

FLOATING-POINT CONSISTENCY

Getting consistent floating-point results when moving to the Intel® Xeon Phi™ x200 processor family from Intel® Xeon® processors or from Intel® Xeon Phi™ x100 Coprocessors

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™ x100 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

...

Math functions

Implementation of math functions may differ between different processors

- **For consistency of math functions between KNL and Intel® Xeon® processors, use**
-fimf-arch-consistency=true for both
- Not available for KNC
 - -fp-model precise (or -fimf-precision=high) should get you close
- These options come at a cost in performance

FMAAs

The most common cause of differences between Intel® Xeon® processors and Intel® Xeon Phi™ x100 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 FMAAs, 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

FMAAs

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
- FMAAs may break the symmetry of an expression:

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

If FMAAs 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.

Other Differences

`-fp-model fast=2` enables some more aggressive optimizations for Intel® MIC™ Architecture

- Faster in-lined versions of some math functions
 - May not give standard behavior for extreme or exceptional arguments
- Assumes smaller dynamic range for complex numbers, so does not protect against overflows, e.g. in complex division (same as on Intel® Xeon® processors)

Can unmask and trap floating-point exceptions on KNL (not KNC)

- `-fpe0 -traceback` (Fortran) or `-fp-trap=common` (C/C++)

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.

Additional Resources

<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://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-17.0-user-and-reference-guide>

<https://software.intel.com/intel-fortran-compiler-17.0-user-and-reference-guide>

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

Intel® Compiler Support at <https://servicetickets.intel.com>

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>

Intel® Compiler Support at <https://servicetickets.intel.com>

Legal Disclaimer & Optimization Notice

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

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

Copyright © 2017, Intel Corporation. All rights reserved. Intel, Xeon, Xeon Phi, Core and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

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

Notice revision #20110804

