Legal Disclaimer & Optimization Notice

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. For more complete information visit [www.intel.com/benchmarks](http://www.intel.com/benchmarks).

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.

Copyright © 2019, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

**Optimization Notice**

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

Notice revision #20110804
Agenda

• Intro
• New Features
• High-Level Optimizations
• Optimization reports
• Interprocedural Optimizations
• Auto-Vectorization
• Profile-Guided Optimizations
• Auto-Parallelization
General Compiler Features

Supports standards
- Full C11, Full C++14, Some C++ 17
  - use -std option to control, e.g. -std=c++14
- Full Fortran 2008, Some F2018

Intel® C/C++ source and binary compatible
- Linux*: gcc
- Windows*: Visual C++ 2013 and above

Supports all instruction sets via vectorization
- Intel® Streaming SIMD Extensions, (Intel® SSE)
- Intel® Advanced Vector Extensions (Intel® AVX)
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512)

Full OpenMP* 4.5 support

Optimized math libraries
- libimf (scalar) and libsvml (vector): faster than GNU libm
- Driver links libimf automatically, ahead of libm
- Replace math.h by mathimf.h

Many advanced optimizations
- With detailed, structured optimization reports

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

The Intel® Xeon® Processor Scalable Family is based on the server microarchitecture codenamed Skylake

- Compile with processor-specific option \([-/Q]xCORE-AVX512\)
- Many HPC apps benefit from aggressive ZMM usage
- Most non-HPC apps degrade from aggressive ZMM usage

A new compiler option \([-q/Q]opt-zmm-usage=low|high\) is added to enable a smooth transition from AVX2 to AVX-512

```c
void foo(double *a, double *b, int size) {
    #pragma omp simd
    for(int i=0; i<size; i++) {
        b[i] = exp(a[i]);
    }
}
```

```bash
icpc -c -xCORE-AVX512 -qopenmp -qopt-report:5 foo.cpp
```

remark #15305: vectorization support: vector length 4
remark #15321: Compiler has chosen to target XMM/YMM vector. Try using \(-qopt-zmm-usage=high\) to override
remark #15478: estimated potential speedup: 5.260
New options and features

The Intel® Xeon® Processor Scalable Family is based on the server microarchitecture codenamed Skylake

- Compile with processor-specific option \([-/Q]xCORE-AVX512\)
- Many HPC apps benefit from aggressive ZMM usage
- Most non-HPC apps degrade from aggressive ZMM usage

A new compiler option \([-q/Q]opt-zmm-usage=low|high\) is added to enable a smooth transition from AVX2 to AVX-512

```c
void foo(double *a, double *b, int size) {
    #pragma omp simd
    for(int i=0; i<size; i++) {
        b[i]=exp(a[i]);
    }
}
```

```bash
icpc -c -xCORE-AVX512 -qopt-zmm-usage=high -qopenmp
-qopt-report:5 foo.cpp
remark #15305: vectorization support: vector length 8
remark #15478: estimated potential speedup: 10.110
```
High-Level Optimizations

Basic Optimizations with icc -O...

- **-00** no optimization; sets -g for debugging
- **-01** scalar optimizations
  excludes optimizations tending to increase code size
- **-02** default for icc/icpc (except with -g)
  includes auto-vectorization; some loop transformations, e.g. unrolling, loop interchange;
inlining within source file;
start with this (after initial debugging at -O0)
- **-03** more aggressive loop optimizations
  including cache blocking, loop fusion, prefetching, ...
suited to applications with loops that do many floating-point calculations or process large data sets
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
## Common optimization options

<table>
<thead>
<tr>
<th>Option</th>
<th>Linux*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable optimization</td>
<td>-O0</td>
</tr>
<tr>
<td>Optimize for speed (no code size increase)</td>
<td>-O1</td>
</tr>
<tr>
<td>Optimize for speed (default)</td>
<td>-O2</td>
</tr>
<tr>
<td>High-level loop optimization</td>
<td>-O3</td>
</tr>
<tr>
<td>Create symbols for debugging</td>
<td>-g</td>
</tr>
<tr>
<td>Multi-file inter-procedural optimization</td>
<td>-ipo</td>
</tr>
<tr>
<td>Profile guided optimization (multi-step build)</td>
<td>-prof-gen</td>
</tr>
<tr>
<td></td>
<td>-prof-use</td>
</tr>
<tr>
<td>Optimize for speed across the entire program (“prototype switch”)</td>
<td>-fast</td>
</tr>
<tr>
<td><strong>fast options definitions changes over time!</strong></td>
<td>same as:</td>
</tr>
<tr>
<td></td>
<td>-ipo -O3 -no-prec-div -static -fp-model fast=2 -xHost)</td>
</tr>
<tr>
<td>OpenMP support</td>
<td>-qopenmp</td>
</tr>
<tr>
<td>Automatic parallelization</td>
<td>-parallel</td>
</tr>
</tbody>
</table>
Compiler Reports – Optimization Report

Enables the optimization report and controls the level of details

- `-qopt-report [=n]`
- When used without parameters, full optimization report is issued on stdout with details level 2

Control destination of optimization report

- `-qopt-report=<filename>`
- By default, without this option, a `<filename>.optrpt` file is generated.

Subset of the optimization report for specific phases only

- `-qopt-report-phase [=list]`
  Phases can be:
  - all  – All possible optimization reports for all phases (default)
  - loop  – Loop nest and memory optimizations
  - vec  – Auto-vectorization and explicit vector programming
  - par  – Auto-parallelization
  - openmp  – Threading using OpenMP
  - ipo  – Interprocedural Optimization, including inlining
  - pgo  – Profile Guided Optimization
  - cg  – Code generation
InterProcedural Optimizations (IPO)

Multi-pass Optimization

\texttt{icc -ipo}

Analysis and optimization across function and/or source file boundaries, e.g.

- Function inlining; constant propagation; dependency analysis; data & code layout; etc.

2-step process:

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

Particularly effective for applications with many smaller functions

Get report on inlined functions with \texttt{-qopt-report-phase=ipo}
## InterProcedural Optimizations

Extends optimizations across file boundaries

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ip</td>
<td>Only between modules of one source file</td>
</tr>
<tr>
<td>-ipo</td>
<td>Modules of multiple files/whole application</td>
</tr>
</tbody>
</table>

### Without IPO

- Compile & Optimize → file1.c
- Compile & Optimize → file2.c
- Compile & Optimize → file3.c
- Compile & Optimize → file4.c

### With IPO

- Compile & Optimize
  - file1.c
  - file3.c
  - file4.c
  - file2.c
Math Libraries

icc comes with Intel's optimized math libraries

- libimf (scalar) and libsvml (scalar & vector)
- Faster than GNU* libm
- Driver links libimf automatically, ahead of libm
- Additional functions (replace math.h by mathimf.h)

Don’t link to libm explicitly!  

- 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
SIMD: **Single Instruction, Multiple Data**

```c
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)

<table>
<thead>
<tr>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

= | =

| X + Y | X + Y |

- 8 doubles for AVX-512
Vectorization

Vectorization is the generation of loops using packed SIMD instructions as above

Different approaches:

- **Auto-vectorization**: The compiler does it all, at -O2 or above
  - **Compiler** is responsible for correctness

- **Assisted vectorization**: User provides additional hints or help via pragmas, language constructs or source code changes

- **Explicit vectorization**: User dictates vectorization using OpenMP* pragmas
  - **User** is responsible for correctness
  - Use when compiler is unable to auto-vectorize or when you want control over how a loop gets vectorized

* Other names and brands may be claimed as the property of others.
Many Ways to Vectorize

- Compiler: Auto-vectorization (no change of code)
- Compiler: Auto-vectorization hints (e.g., #pragma vector, ...)
- Compiler: OpenMP* SIMD directives
  - SIMD intrinsic class (e.g., F32vec, F64vec, ...)
  - Vector intrinsic (e.g., _mm_fmadd_pd(...), _mm_add_ps(...), ...)
  - Assembler code (e.g., [v]addps, [v]addss, ...)

Ease of use

Programmer control
## Supported Processor-Specific Compiler Switches

<table>
<thead>
<tr>
<th>Intel® processors only</th>
<th>Intel and non-Intel (-m also GCC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-xsse2</td>
<td>-msse2 (default)</td>
</tr>
<tr>
<td>-xsse3</td>
<td>-msse3</td>
</tr>
<tr>
<td>-xssse3</td>
<td>-mssse3</td>
</tr>
<tr>
<td>-xsse4.1</td>
<td>-msse4.1</td>
</tr>
<tr>
<td>-xsse4.2</td>
<td>-msse4.2</td>
</tr>
<tr>
<td>-xavx</td>
<td>-mavx</td>
</tr>
<tr>
<td>-xcore-avx2</td>
<td></td>
</tr>
<tr>
<td>-xcore-avx512</td>
<td></td>
</tr>
<tr>
<td>-xHost</td>
<td>-xHost (-march=native)</td>
</tr>
</tbody>
</table>

**Intel cpuid check**
- No cpu id check

**Runtime message if run on unsupported processor**
- Illegal instruction error if run on unsupported processor
Dependencies – An Obstacle to Vectorization

Vectorization is safe only if one SIMD lane doesn’t depend on the result of another (previous) one.

```
for (int i = 0; i < N; i++)
a[i + 2] = a[i] + c;
```

In this example, the result for \( i=2, a[4] \), depends on the result for \( i=0, a[2] \).

This is a “loop-carried”, “vector” or “flow” dependency. If both \( a[4] \) and \( a[2] \) are computed in the same SIMD instruction, the result for \( a[2] \) is not yet available when \( a[4] \) is computed.

The original value for \( a[2] \) will be used and the result for \( a[4] \) will be incorrect.

The compiler won’t auto-vectorize this loop and you mustn’t force it to.
The Compiler Must Be Conservative

Not all potential dependencies are real.

- The compiler assumes that pointers `a` and `b` might be “aliased”
- Loop multi-versioning for Data Dependence may help
- You can help the compiler to vectorize
  - compile with `-fargument-noalias`
  - use the “restrict” keyword: `void scale(int *a, int *restrict b)`
  - insert `#pragma ivdep` before the for loop

```c
void scale(int *a, int *b)
{
    for (int i = 0; i < 10000; i++) b[i] = a[i] + 4;
}
```
Guidelines for Writing Vectorizable Code

- **Prefer simple “for” loops** (trip count should be known at loop entry)
- **Avoid dependencies on an earlier loop iteration**
- **Write straight line code.** Avoid:
  - Most function calls (unless inlined or simple math functions)
  - Branches that can’t be treated as masked assignments.
- **Prefer arrays to the use of pointers**
  - Without help, the compiler often can’t tell if 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 (contiguous memory access)
  - Minimize indirect addressing, such as \( b[i] = a[index[i]] \)
  - Align your data consistently where possible, to 64 byte boundaries (for Intel® AVX-512)
Reasons Loops May Not Be Auto-Vectorized

- Trip count (number of iterations) not known at loop entry
- Dependency on an earlier iteration (explicit or potential through pointer aliasing)
  - Function calls inside loop
- Compiler doesn’t expect performance gain
  - Non-linear or non-contiguous memory access
  - Low trip count (number of loop iterations) – compare to number of SIMD lanes
- Complex control flow
  - large switch statements; goto statements; exception handling; many nested “if”s
- Complex (complicated) data types with user-defined operators
- Outer loop of loop nest
Optimization Report – An Example

$icc -c -xcommon-avx512 -qopt-report=3 -qopt-report-phase=loop,vec foo.c

Creates foo.optrpt summarizing which optimizations the compiler performed or tried to perform. Level of detail from 0 (no report) to 5 (maximum).
-qopt-report-phase=loop,vec asks for a report on vectorization and loop optimizations only

Extracts:

LOOP BEGIN at foo.c(4,3)

Multiversioned v1
remark #25228: Loop multiversioned for Data Dependence...
remark #15300: LOOP WAS VECTORIZED
remark #15450: unmasked unaligned unit stride loads: 1
remark #15451: unmasked unaligned unit stride stores: 1
.... (loop cost summary) ....
LOOP END

LOOP BEGIN at foo.c(4,3)

<Multiversioned v2>
remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning
LOOP END

#include <math.h>
void foo (float * theta, float * sth) {
  int i;
  for (i = 0; i < 512; i++)
    sth[i] = sin(theta[i]+3.1415927);
}
Optimization Report – An Example

```
...
LOOP BEGIN at foo.c(4,3)
...
remark #15417: vectorization support: number of FP up converts: single precision to double precision 1 [ foo.c(5,17) ]
remark #15418: vectorization support: number of FP down converts: double precision to single precision 1 [ foo.c(5,8) ]
remark #15300: LOOP WAS VECTORIZED
remark #15450: unmasked unaligned unit stride loads: 1
remark #15451: unmasked unaligned unit stride stores: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 111
remark #15477: vector cost: 10.310
remark #15478: estimated potential speedup: 10.740
remark #15482: vectorized math library calls: 1
remark #15487: type converts: 2
remark #15488: --- end vector cost summary ---
remark #25015: 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 < 512; i++)
        sth[i] = sin(theta[i]+3.1415927);
}
```
Optimization Report – An Example


LOOP BEGIN at foo2.c(4,3)
...
remark #15305: vectorization support: vector length 32
remark #15300: LOOP WAS VECTORIZED
  remark #15450: unmasked unaligned unit stride loads: 1
  remark #15451: unmasked unaligned unit stride stores: 1
remark #15475: --- begin vector cost summary ---
remark #15476: scalar cost: 109
remark #15477: vector cost: 5.250
remark #15478: estimated potential speedup: 20.700
remark #15482: vectorized math library calls: 1
remark #15488: --- end vector cost summary ---
remark #25015: Estimate of max trip count of loop=32
LOOP END

$grep sin foo.s
  call __svml_sinf16_b3
  call __svml_sinf16_b3

#include <math.h>
void foo (float * theta, float * sth) {
    int i;
    for (i = 0; i < 512; i++)
        sth[i] = sinf(theta[i]+3.1415927f);
}
Validating Vectorization Success I

Optimization report:

- Linux*, macOS*: `-qopt-report=<n>`, Windows*: `/Qopt-report:<n>
  n: 0, ..., 5 specifies level of detail; 2 is default (more later)
- Prints optimization report with vectorization analysis

Optimization report phase:

- Linux*, macOS*: `-qopt-report-phase=<p>`,
  Windows*: `/Qopt-report-phase:<p>
- `<p>` is all by default; use vec for just the vectorization report

Optimization report file:

- `<f>` can be stderr, stdout or a file (default: *.optrpt)
Validating Vectorization Success II

Assembler code inspection (Linux*, macOS*: -S, Windows*: /Fa):

- Most reliable way and gives all details of course
- Check for scalar/packed or (E)VEX encoded instructions:
  Assembler listing contains source line numbers for easier navigation
- Compiling with -qopt-report-embed (Linux*, macOS*) or /Qopt-report-embed (Windows*) helps interpret assembly code

Performance validation:

- Compile and benchmark with -no-vec -qno-openmp-simd or /Qvec- /Qopenmp-simd-, or on a loop by loop basis via
  #pragma novector or !DIR$ NOVECTOR
- Compile and benchmark with selected SIMD feature
- Compare runtime differences

Intel® Advisor
Profile-Guided Optimizations (PGO)

Static analysis leaves many questions open for the optimizer like:
- How often is $x > y$
- What is the size of count
- Which code is touched how often

Use execution-time feedback to guide (final) optimization

Enhancements with PGO:
- More accurate branch prediction
- Basic block movement to improve instruction cache behavior
- Better decision of functions to inline (help IPO)
- Can optimize function ordering
- Switch-statement optimization
- Better vectorization decisions

```c
if (x > y)
    do_this();
else
    do_that();
```

```c
for(i=0; i<count; ++i)
    do_work();
```
PGO Usage: Three-Step Process

Step 1
Compile + link to add instrumentation
icc -prof-gen prog.c -o prog

Instrumented executable:
prog

Step 2
Execute instrumented program
./prog (on a typical dataset)

Dynamic profile:
12345678.dyn

Step 3
Compile + link using feedback
icc -prof-use prog.c -o prog

Merged .dyn files:
pgopti.dpi

Optimized executable:
prog
Auto-Parallelization

Based on OpenMP* runtime

Compiler automatically translates loops into equivalent multithreaded code with using this option: 
-parallel

The auto-parallelizer detects simply structured loops that may be safely executed in parallel and automatically generates multi-threaded code for these loops.

The auto-parallelizer report can provide information about program sections that were parallelized by the compiler. Compiler switch: 
-qopt-report-phase=par

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

- Binary floating-point [FP] representations of most real numbers are inexact, and there is an inherent uncertainty in the result of most calculations involving floating-point numbers.

- Programmers of floating-point applications typically have the following objectives:
  - **Accuracy**: Produce results that are “close” to the correct value.
  - **Reproducibility**: Produce consistent results:
    - From one run to the next
    - From one set of build options to another
    - From one compiler to another
    - From one platform to another
  - **Performance**: Produce the most efficient code possible.
Floating Point Consistency

- Binary floating-point [FP] representations of most real numbers are inexact, and there is an inherent uncertainty in the result of most calculations involving floating-point numbers.

- Programmers of floating-point applications typically have the following objectives:
  - **Accuracy**: Produce results that are “close” to the correct value
  - **Reproducibility**: Produce consistent results
    - From one run to the next
    - From one set of build options to another
    - From one compiler to another
    - From one platform to another
  - **Performance**: Produce the most efficient code possible

These options usually conflict!
Why Results Vary I

Basic problem:
- FP numbers have **finite resolution** and
- **Rounding** is done for each (intermediate) result

Caused by algorithm:
Conditional numerical computation for different systems and/or input data can have unexpected results

Non-deterministic task/thread scheduler:
Asynchronous task/thread scheduling has best performance but reruns use different threads

Alignment (heap & stack):
If alignment is not guaranteed and changes between reruns the data sets could be computed differently (e.g. vector loop prologue & epilogue of unaligned data)

⇒ User controls those (direct or indirect)
Why Results Vary II

Order of FP operations has impact on rounded result, e.g.

\[(a+b)+c \neq a+(b+c)\]

\[2^{-63} + 1 + -1 = 2^{-63}\] (mathematical result)

\[(2^{-63} + 1) + -1 \approx 0\] (correct IEEE result)

\[2^{-63} + (1 + -1) \approx 2^{-63}\] (correct IEEE result)

Constant folding: \(X + 0 \Rightarrow X\) or \(X \times 1 \Rightarrow X\)

Multiply by reciprocal: \(A/B \Rightarrow A \times (1/B)\)

Approximated transcendental functions (e.g. \(\text{sqrt}(...)\), \(\text{sin}(...)\), ...)

Flush-to-zero (for SIMD instructions)

Contractions (e.g. FMA)

Different code paths (e.g. SIMD & non-SIMD or Intel AVX vs. SSE)

...

⇒ Subject of Optimizations by Compiler & Libraries
Compiler Optimizations

Why compiler optimizations:
- Provide best performance
- Make use of processor features like SIMD (vectorization)
- In most cases performance is more important than FP precision and reproducibility
- Use faster FP operations (not legacy x87 coprocessor)

FP model of compiler limits optimizations and provides control about FP precision and reproducibility:

Default is "fast"

Controlled via:
Linux*: –fp-model
FP Model

FP model settings:

- **precise**: allows value-safe optimizations only
- **source/double/extended**: intermediate precision for FP expression eval.
- **except**: enables strict floating point exception semantics
- **strict**: enables access to the FPU environment disables floating point contractions such as fused multiply-add (fma) instructions implies “precise” and “except”
- **fast [=1]** (default):
  Allows value-unsafe optimizations compiler chooses precision for expression evaluation
  Floating-point exception semantics not enforced
  Access to the FPU environment not allowed
  Floating-point contractions are allowed
- **fast=2**: some additional approximations allowed
- **consistent**: consistent, reproducible floating-point results for different optimization levels or between different processors of the same architecture
OPENMP* DETERMINISTIC REDUCTION

- **KMP_DETERMINISTIC_REDUCTION:** Enables (1) or disables (0) the use of a specific ordering of the reduction operations for implementing the reduction clause for an OpenMP* parallel region. This has the effect that, for a given number of threads, in a given parallel region, for a given data set and reduction operation, a floating point reduction done for an OpenMP reduction clause will have a consistent floating point result from run to run, since round-off errors will be identical.

- Use with –fp-model [precise|strict]!
Linear algebra, FFTs, sparse solvers, statistical, ...

- Highly optimized, vectorized
- Threaded internally using OpenMP*
- By default, repeated runs may not give identical results

Conditional Numerical Reproducibility

- Repeated runs give identical results under certain conditions:
  - Same number of threads
  - `OMP_SCHEDULE=static` (the default)
  - Same OS and architecture (e.g. Intel 64)
  - Same microarchitecture, or specify a minimum microarchitecture

- Call `mkl_cbwr_set(MKL_CBWR_AUTO)` (run-to-run)
- Call `mkl_cbwr_set(MKL_CBWR_COMPATIBLE)` (processors)
- Or set environment variable `MKL_CBWR=...` (etc.)

## Controls for CNR Features

<table>
<thead>
<tr>
<th>For consistent results ...</th>
<th>Function Call</th>
<th>Environment Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>on Intel® or Intel®-compatible CPUs supporting SSE2 instructions or later</td>
<td>MKL_CBWR_SET( ... )</td>
<td>MKL_CBWR_COMPATIBLE</td>
</tr>
<tr>
<td>on Intel® processors supporting SSE2 instructions or later</td>
<td>MKL_CBWR_SSE2</td>
<td>COMPATIBLE</td>
</tr>
<tr>
<td>on Intel processors supporting SSE4.2 instructions or later</td>
<td>MKL_CBWR_SSE4_2</td>
<td>SSE2</td>
</tr>
<tr>
<td>on Intel processors supporting Intel® AVX or later</td>
<td>MKL_CBWR_AVX</td>
<td>SSE4_2</td>
</tr>
<tr>
<td>from run to run (but not processor-to-processor)</td>
<td>MKL_CBWR_AUTO</td>
<td>AUTO</td>
</tr>
</tbody>
</table>
program rep
  use mpi
  implicit none
  integer :: n_ranks, rank, errc
  real*8 :: global_sum, local_value

  call MPI_Init(errc)
  call MPI_Comm_size(MPI_COMM_WORLD, n_ranks, errc)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, errc)

  local_value = 2.0 ** -60

  if(rank.eq.15) local_value = +1.0
  if(rank.eq.16) local_value = -1.0

  call MPI_Reduce(local_value, global_sum, 1, MPI_DOUBLE_PRECISION, &
  MPI_SUM, 0, MPI_COMM_WORLD, errc)

  if(rank.eq.0) write(*,'(f22.20)') global_sum

  call MPI_Finalize(errc)
end program rep

$ cat ${machinefile_A}
ehk248:16
ehs146:16
ehs231:16
ehs145:16
$ cat ${machinefile_B}
ehk248:32
ehs146:32
ehs231:0
ehs145:0
$ mpiifort -fp-model strict -o ./rep.x ./rep.f90

$ export I_MPI_ADJUST_REDUCE=3
$ mpirun -n 64 -machinefile ${machinefile_A} ./rep.x
0.00000000000000000000
$ mpirun -n 64 -machinefile ${machinefile_B} ./rep.x
0.00000000000000004163

$ export I_MPI_ADJUST_REDUCE=1
$ mpirun -n 64 -machinefile ${machinefile_A} ./rep.x
0.00000000000000000000
$ mpirun -n 64 -machinefile ${machinefile_B} ./rep.x
0.00000000000000004163
Backup
Alignment

Caveat with using unaligned memory access:

- Unaligned loads and stores can be **very slow** due to higher I/O because two cache-lines need to be loaded/stored (not always, though)

- Compiler can mitigate expensive unaligned memory operations by using two partial loads/stores – **still slow**
  
  (e.g. two 64 bit loads instead of one 128 bit unaligned load)

- The compiler can use “versioning” in case alignment is unclear:
  
  Run time checks for alignment to use fast aligned operations if possible, the slower operations otherwise – **better but limited**

**Best performance:** User defined aligned memory

- 16 byte for SSE
- 32 byte for AVX
- 64 byte for Intel® MIC Architecture & Intel® AVX-512
Alignment Hints for C/C++ I

- Aligned heap memory allocation by intrinsic/library call:
  - `void* _mm_malloc(int size, int base)`
  - Linux* only: `int posix_memaligned(void **p, size_t base, size_t size)`

- `#pragma vector [aligned|unaligned]`
  - Only for Intel Compiler
  - Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  - Use with care:
    The assertion must be satisfied for all(!) data accesses in the loop!
Alignment Hints for C/C++ II

- Align attribute for variable declarations:
  - __declspec(align(base)) <var>
  - Only Linux*: <var> __attribute__((aligned(base)))
  - Portability caveat: __declspec is not known for GCC and __attribute__ not for Microsoft Visual Studio*!

- Hint that start address of an array is aligned (Intel Compiler only):
  __assume_aligned(<array>, base)

- alignas specifier (since C++11):
  alignas( expression )
Alignment Hints for Fortran

• !DIR$ VECTOR [ALIGNED|UNALIGNED]
  ▪ Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  ▪ **Use with care:**
    The assertion must be satisfied for all(!) data accesses in the loop!

• Hint that an entity in memory is aligned:
  !DIR$ ASSUME_ALIGNED address1:base [, address2:base] ...

• Align variables:
  !DIR$ ATTRIBUTES ALIGN: base :: variable

• Align data items globally:
  Linux*: -align <a>
  ▪ <a> can be array<n>byte with <n> defining the alignment for arrays
  ▪ Other values for <a> are also possible, e.g.: [no]commons, [no]records, ...
Pragma SIMD

- **Pragma SIMD:**
  The `simd` construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions). [OpenMP* 4.0 API: 2.8.1]

**Syntax:**

- **C/C++:**
  ```
  #pragma omp simd [clause [,clause]...]
  ```

- **Fortran:**
  ```
  !DIR$ OMP SIMD [clause [,clause]...]
  ```

- **For-loop has to be in “canonical loop form”** (see OpenMP 4.0 API:2.6)
  - Random access iterators required for induction variable (integer types or pointers for C++)
  - Limited test and in-/decrement for induction variable
  - Iteration count known before execution of loop
Pragma SIMD Clauses

- **safelen(n1[,n2] ...)**: \(n1, n2, \ldots\) must be power of 2: The compiler can assume a vectorization for a vector length of \(n1, n2, \ldots\) to be safe

- **private(v1, v2, ...)**: Variables private to each iteration
  - **lastprivate(...)**: last value is copied out from the last iteration instance

- **linear(v1:step1, v2:step2, ...)**: For every iteration of original scalar loop \(v1\) is incremented by \(step1, \ldots\) etc. Therefore it is incremented by \(step1 \times \text{vector length}\) for the vectorized loop.

- **reduction(operator:v1, v2, ...)**: Variables \(v1, v2, \ldots\) etc. are reduction variables for operation \(operator\)

- **collapse(n)**: Combine nested loops – collapse them

- **aligned(v1:base, v2:base, ...)**: Tell variables \(v1, v2, \ldots\) are aligned; (default is architecture specific alignment)
Pragma SIMD Example

Ignore data dependencies, indirectly mitigate control flow dependence & assert alignment:

```c
void vec1(float *a, float *b, int off, int len)
{
    #pragma omp simd safelen(32) aligned(a:64, b:64)
    for(int i = 0; i < len; i++)
    {
        a[i] = (a[i] > 1.0) ?
            a[i] * b[i] :
            a[i + off] * b[i];
    }
}
```

LOOP BEGIN at simd.cpp(4,5)
remark #15388: vectorization support: reference a has aligned access   [ simd.cpp(6,9) ]
remark #15388: vectorization support: reference b has aligned access   [ simd.cpp(6,9) ]
...
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
...
LOOP END
SIMD-Enabled Functions

- **SIMD-Enabled Function (aka. declare simd construct):**
  The declare simd construct can be applied to a function [...] to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.
  [OpenMP* 4.0 API: 2.8.2]

- **Syntax:**

  **C/C++:**
  ```
  #pragma omp declare simd [clause [,clause]...]  
  ```

  **Fortran:**
  ```
  !$OMP DECLARE SIMD [clause[,[,] clause]...]  
  ```

- **Intent:**

  Express work as scalar operations (kernel) and let compiler create a vector version of it. The size of vectors can be specified at compile-time (SSE, AVX, …) which makes it portable!
SIMD-Enabled Function Clauses

- **simdlen(len)**
  len must be power of 2: Allow as many elements per argument (default is implementation specific)

- **linear(v1:step1, v2:step2, ...)**
  Defines v1, v2, ... to be private to SIMD lane and to have linear (step1, step2, ...) relationship when used in context of a loop

- **uniform(a1, a2, ...)**
  Arguments a1, a2, ... etc. are not treated as vectors (constant values across SIMD lanes)

- **inbranch, notinbranch**: SIMD-enabled function called only inside branches or never

- **aligned(a1:base, a2:base, ...)**: Tell arguments a1, a2, ... are aligned; (default is architecture specific alignment)
SIMD-Enabled Function Example

Ignore data dependencies, indirectly mitigate control flow dependence & assert alignment:

```c
#pragma omp declare simd simdlen(16)notinbranch uniform(a, b, off)
float work(float *a, float *b, int i, int off)
{
    return (a[i] > 1.0) ? a[i] * b[i] : a[i + off] * b[i];
}

void vec2(float *a, float *b, int off, int len)
{
    #pragma omp simd safelen(64) aligned(a:64, b:64)
    for(int i = 0; i < len; i++)
        a[i] = work(a, b, i, off);
}
```

INLINE REPORT: (vec2(float *, float *, int, int)) [4/9=44.4%] simd.cpp(8,1)
    -> INLINE: (12,16) work(float *, float *, int, int) (isz = 18) (sz = 31)

LOOP BEGIN at simd.cpp(10,5)
    remark #15388: vectorization support: reference a has aligned access [ simd.cpp(4,20) ]
    remark #15388: vectorization support: reference b has aligned access [ simd.cpp(4,20) ]
    ...
    remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
    ...
LOOP END
Annotated Source Listing

Get reports as annotation to source files:

- Linux*, macOS*: `-qopt-report-annotate=[text|html]`,
  Windows*: `/Qopt-report-annotate=[text|html]`

- *.annot file is generated

```c
// ------ Annotated listing with optimization reports for "test.cpp" -------
// 1   void add(double *A, double *B, double *C, double *D)  
2     {  
3       for (int i = 0; i < 1000; i++)  
...  
//LOOP BEGIN at test.cpp(3,2)  
//Multiversioned v1  
//test.cpp(3,2):remark #15300: LOOP WAS VECTORIZED  
//LOOP END  
...  
4       D[i] = A[i] + B[i]+C[i];  
5     }  
6
```
Tuning for Skylake - Compiler options

Both Skylake and Knights Landing processors have support for Intel® AVX-512 instructions. There are three ISA options in the Intel® Compiler:

- `-xCORE-AVX512` : Targets Skylake, contains instructions not supported by Knights Landing
- `-xCOMMON-AVX512` : Targets both Skylake and Knights Landing
- `-xMIC-AVX512` : Targets Knights Landing, includes instructions not supported by Skylake

Intel® Compiler is conservative in its use of ZMM (512bit) registers so to enable their use with Skylake the additional flag `-qopt-zmm-usage=high` must be set.