Intel® COMPOSER XE Tips

Presenter: Kenneth Craft

Date: 03-09-2017

Agenda

Introduction

Optimizations and Reports

Floating point Model

OpenMP SIMD

KNL Memory and Floating Point

Why Use Intel® Compilers?

Compatibility

- Multiple OS Support: Windows*, Linux*, OS X*
- Integration into development environments: Visual Studio* in Windows*, Eclipse* in Linux*, Xcode* in OS X*
- Compatible with most versions of the GNU* Compiler collection (gcc)
- Source and binary compatibility with Microsoft Visual C++* Compilers
- Most features from C99 Standard (C Compilers)
- Full C++11 Standard support, some features from C++14 supported (C++ Compilers)
- Fortran 2003, Many features from Fortran 2008
- Support for Draft Fortran 2015 features

Parallelism

- Explicit Vector Programming (OpenMP*)
- Extensive OpenMP* 4.1 support
- C++ Multithreading Library (Intel® TBB)

Why Use Intel® Compilers?

Performance

- Code generation tuned for latest microarchitecture
- New instructions enable new opportunities (SSE, AVX, AVX2, AVX-512)
- Domain specific performance libraries (Intel® MKL, Intel® IPP)
- Data analytics acceleration library (Intel® DAAL)

Optimization

- Optimizing compilers
- Automatic vectorization
- Intel's optimized version of libm (Intel® Math Library libimf)

Common Optimization Options

	Windows*	Linux*, OS X*
Disable optimization	/Od	-O0
Optimize for speed (no code size increase)	/O1	-O1
Optimize for speed (default)	/O2	-O2
High-level loop optimization	/O3	-O3
Create symbols for debugging	/Zi	-g
Multi-file inter-procedural optimization	/Qipo	-ipo
Profile guided optimization (multi-step build)	/Qprof-gen /Qprof-use	-prof-gen -prof-use
Optimize for speed across the entire program ("prototype switch") ***fast* options definitions changes over time!**	/fast same as: /O3 /Qipo / Qprec-div-, / fp:fast=2 /QxHost)	-fast same as: Linux: -ipo –O3 -no-prec-div –static –fp-model fast=2 -xHost) OS X: -ipo -mdynamic-no-pic -O3 -no-prec-div -fp-model fast=2 -xHost
OpenMP support	/Qopenmp	-qopenmp
Automatic parallelization	/Qparallel	-parallel

Compiler Reports – Optimization Report

- Enables the optimization report and controls the level of details
  - `/Qopt-report[:n], -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-file:<filename>, -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], -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
    - offload – offload of data and/or execution to Intel® MIC Architecture or to Intel® Graphics Technology
      Note: “offload” does not report on optimizations for MIC, it reports on data that are offloaded.

High-Level Optimization (HLO)

Compiler switches:
`/O2, -O2` **(default)**, `/O3, -O3`

- O3 is suited to applications that have loops that do many floating-point calculations or process large data sets.
- Some of the optimizations are the same as at O2, but are carried out more aggressively. Some poorly suited applications might run slower at O3 than O2

Loop level optimizations

- loop unrolling, cache blocking, prefetching

More aggressive dependency analysis

- Determines whether or not it's safe to reorder or parallelize statements

Scalar replacement

- Goal is to reduce memory by replacing with register references

Interprocedural Optimizations (IPO)
Multi-pass Optimization

- Interprocedural optimizations performs a static, topological analysis of your application!
- ip: Enables inter-procedural optimizations for current source file compilation
- ipo: Enables inter-procedural optimizations across files
  - Can inline functions in separate files
  - Especially many small utility functions benefit from IPO

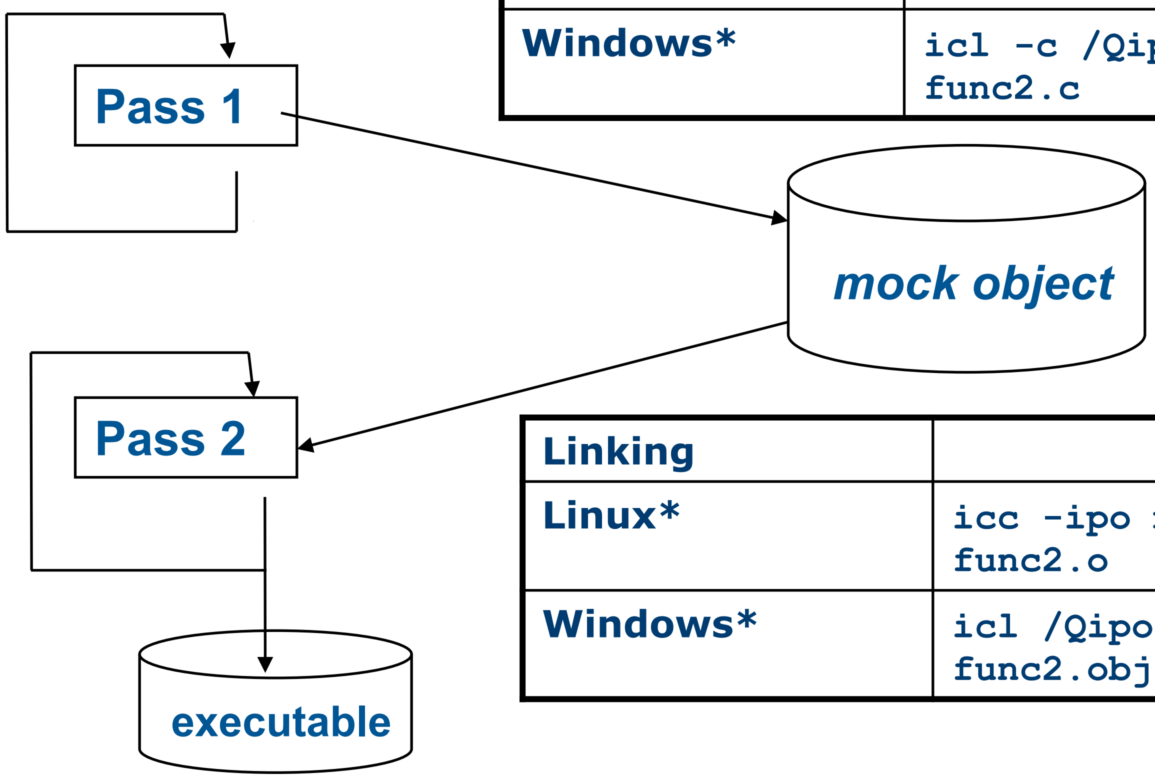
Windows*	Linux*
/Qip	-ip
/Qipo	-ipo

Enabled optimizations:

- Procedure inlining (reduced function call overhead)
- Interprocedural dead code elimination, constant propagation and procedure reordering
- Enhances optimization when used in combination with other compiler features
- Much of ip (including inlining) is enabled by default at option O2

Interprocedural Optimizations (IPO)
Usage: Two-Step Process

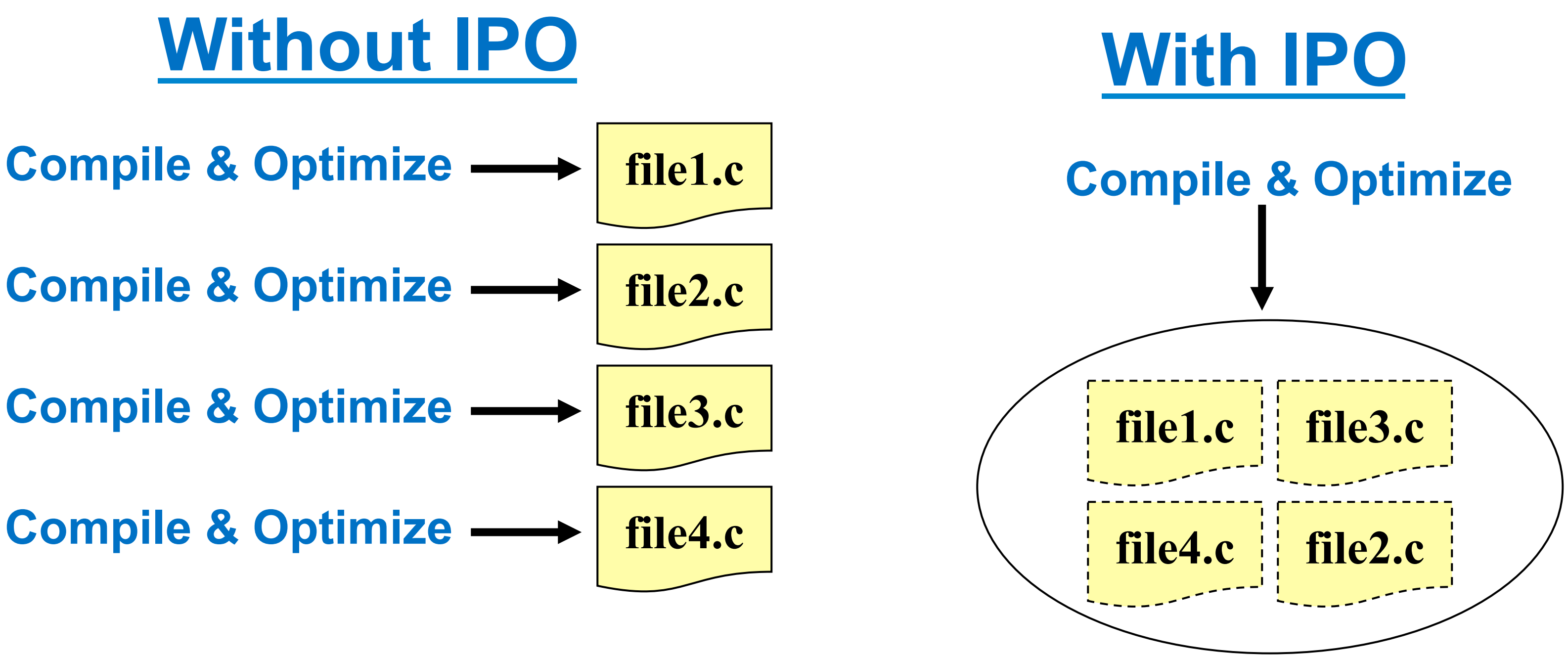
Compiling	
Linux*	`icc -c -ipo main.c func1.c func2.c`
Windows*	`icl -c /Qipo main.c func1.c func2.c`



Linking	
Linux*	`icc -ipo main.o func1.o func2.o`
Windows*	`icl /Qipo main.o func1.o func2.obj`

Interprocedural Optimizations

Extends optimizations across file boundaries



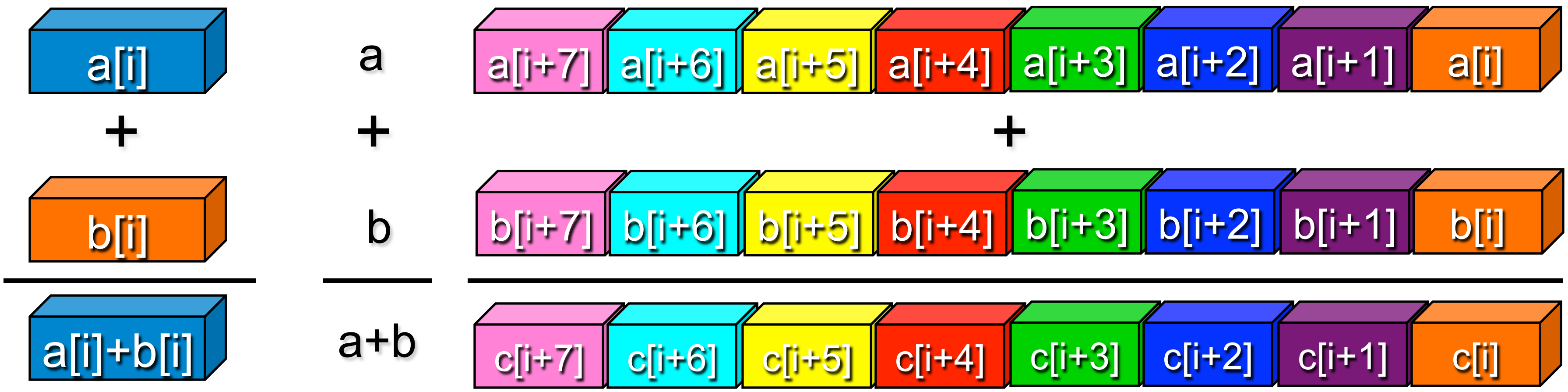
`/Qip, -ip`	Only between modules of one source file
`/Qipo, -ipo`	Modules of multiple files/whole application

Auto-Vectorization

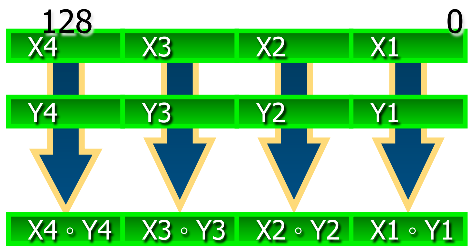
SIMD – Single Instruction Multiple Data

- Scalar mode
  - one instruction produces one result
- SIMD processing
  - with SSE or AVX instructions
  - one instruction can produce multiple results

```
for (i=0;i<=MAX;i++)
     c[i]=a[i]+b[i];
```



Vectorization is Achieved through SIMD Instructions & Hardware



Intel® SSE

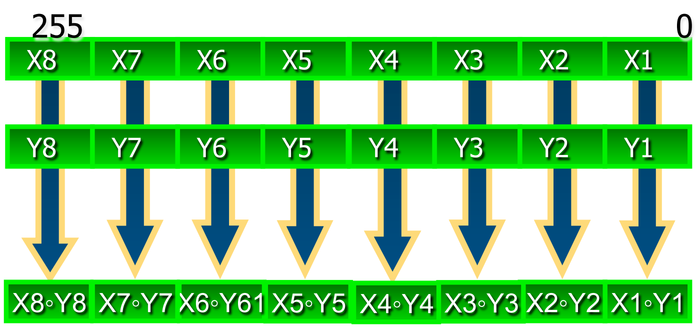
Vector size: 128bit

Data types:

  8,16,32,64 bit integers

  32 and 64bit floats

VL: 2, 4, 8, 16



Intel® AVX

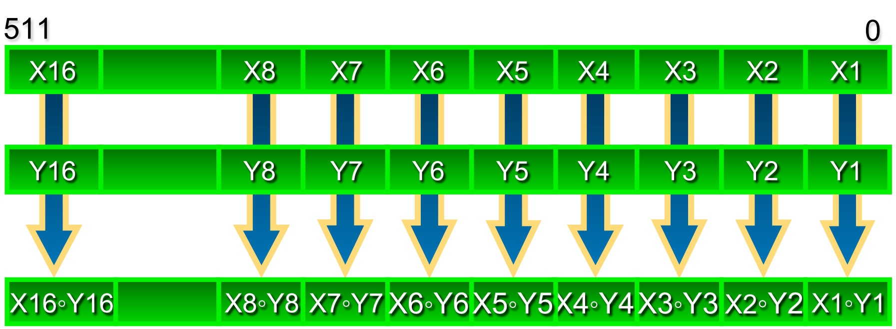
Vector size: 256bit

Data types:

  8, 16, 32, 64 bit integer

  32 and 64 bit float

VL: 4, 8, 16, 32



Intel® AVX-512, Intel® MIC Architecture

Vector size: 512bit

Data types:

  32bit integer

  32 and 64 bit float

VL: 8, 16   Xi, Yi & results 32-bit integer

Automatic Vectorization by Compiler

Intel Compiler will auto vectorize the source code for you if it can

Pros:

- Minimal effort required
- Maintainable – source code is not changed
- Portable across Intel SIMD architectures
- Optimal performance is possible in best cases
- Scales forward!

Cons:

- Compiler is conservative; will not risk generating code that could possibly be unsafe

  => Advanced optimization techniques help to improve Data Level Parallelization using Vectorization

More Vectorization details in a separate training module

Processor Specific Optimizations

Processor specific extensions switches:

`/arch:<target>` (Microsoft compatible), `-m<target>` (Linux*, OS X*)

- No Intel processor check
- No Intel specific optimizations

`/Qx<target>`, `-x<target>`

- Intel specific optimizations
- Processor-check added to main-program

`/Qax<target>`, `-ax<target>`

- Intel specific optimizations
- Autodispatch switch **a** for generating one or more additional optimized code paths

`/QxHost, -xHost`

- Generates optimized code targeted for execution on the system you compile on

Compiler Based Vectorization
Extension Specification

Feature	SIMD Extension
May generate Intel® Streaming SIMD Extensions 2 (Intel® SSE2) instructions as available in initial Pentium® 4 or compatible non-Intel processors	SSE2
May generate Intel® Streaming SIMD Extensions 3 (Intel® SSE3) instructions as available in Pentium® 4 or compatible non-Intel processors	SSE3
May generate Supplemental Streaming SIMD Extensions 3 (SSSE3) instructions as available in Intel® Core™2 Duo processors	SSSE3
May generate Intel® SSE4.1 instructions as first introduced in Intel® 45nm Hi-K next generation Intel Core™ micro-architecture	SSE4.1
May generate Intel® SSE4.2 Accelerated String and Text Processing instructions as available in the previous generation Intel® Core™ processor family	SSE4.2
Like SSSE3 (or SSE4.2) but optimizes for the Intel® Atom™ processors which support SSSE3 (or SSE4.2)	ATOM_SSSE3 (ATOM_SSE4.2)
May generate Intel® Advanced Vector Extensions (Intel® AVX) as available in 2nd generation Intel® Core™ processor family	AVX
May generate Intel® Advanced Vector Extension (Intel® AVX) instructions including instructions offered by the 3rd generation Intel® Core processor	CORE-AVX-I

Compiler Based Vectorization
Extension Specification cont'd

Feature	SIMD Extension
May generate Intel® Advanced Vector Extension (Intel® AVX) instructions including instructions offered by the 3rd generation Intel® Core processor	CORE-AVX-I
May generate Intel® Advanced Vector Extension 2 (Intel® AVX2) instructions as available in the 4th generation Intel® Core™ processor family	CORE-AVX2
May generate Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Foundation, Conflict Detection, Exponential/Reciprocal and Prefetch instructions for Intel® processors, and the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.	MIC-AVX512
May generate Intel® AVX-512 Foundation, Conflict Detection, Doubleword and Quadword, Byte and Word instructions and Intel® AVX-512 Vector Length Extensions for Intel® processors, and the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.	CORE-AVX512
May generate Intel® AVX-512 Foundation, Conflict Detection instructions, as well as the instructions enabled with CORE-AVX2. Optimizes for Intel® processors that support Intel® AVX-512 instructions.	COMMON-AVX512

Example of New Optimization Report

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

Begin optimization report for: foo

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

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

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

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

Optimization Report Example

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

(/Qalias-args- on Windows*)

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

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

Optimization Report Example

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

Begin optimization report for: foo

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

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

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

Optimization Report Example

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

Begin report for: foo

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

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

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

Optimization Report Example

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

Begin optimization report for: foo

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

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

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

Optimization Report Example

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

Begin optimization report for: foo

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

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

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

Compiler Floating-Point (FP) Model

The Floating Point options allow to control the optimizations on floating-point data. These options can be used to tune the performance, level of accuracy or result consistency.

Accuracy

Produce results that are “close” to the correct value

- Measured in relative error, possibly ulps (units in the last place)

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

- Default, primary goal of Intel® Compilers

These objectives usually conflict! Wise use of compiler options lets you control the tradeoffs.

Compiler Floating-Point Model

Problem Statement

Numerical FP results change from run-to-run:

```
C:\Users\me>test.exe
4.012345678901111

C:\Users\me>test.exe
4.012345678902222
```

≠

Numerical results change between different systems:

Intel® Xeon® Processor E5540

```
C:\Users\me>test.exe
4.012345678901111

C:\Users\me>test.exe
4.012345678901111
```

≠

Intel® Xeon® Processor E3-1275

```
C:\Users\me>test.exe
4.012345678902222

C:\Users\me>test.exe
4.012345678902222
```

Why Reproducible FP Results?

Technical/legacy
Software correctness is determined by comparison to previous (baseline) results.

Debugging/porting
When developing and debugging, a higher degree of run-to-run stability is required to find potential problems.

Legal
Accreditation or approval of software might require exact reproduction of previously defined results.

Customer perception
Developers may understand the technical issues with reproducibility but still require reproducible results since end users or customers will be disconcerted by the inconsistencies.

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 ≠ 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` ⇨ `X` or `X * 1` ⇨ `X`

Multiply by reciprocal: `A/B` ⇨ `A * (1/B)`

Approximated transcendental functions (e.g. `sqrt(…)`, `sin(…)`, …)

Flush-to-zero (for SIMD instructions)

Contractions (e.g. FMA)

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

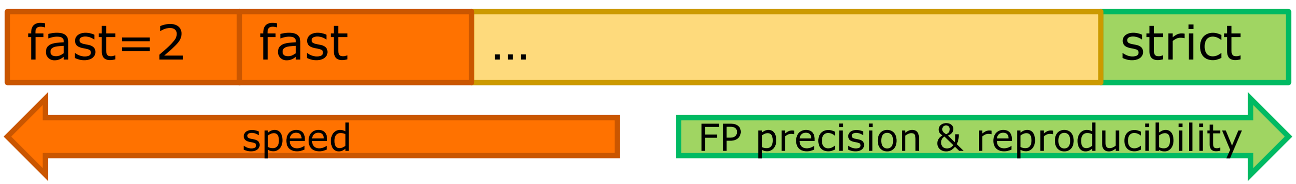
…

⇨ **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*, OS X*: **`–fp-model`**
Windows*: **`/fp:`**

FP Model I

FP model does more:

- Value safety
- Floating-point expression evaluation
- Precise floating-point exceptions
- Floating-point contractions
- Floating-point unit (FPU) environment access

FP Model II

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

FP Model - Comparison

Key	Value Safety	Expression Evaluation	FPU Environ. Access	Precise FP Exceptions	FP contract
precise source double extended	Safe	Varies Source Double Extended	No	No	Yes
strict	Safe	Varies	Yes	Yes	No
fast=1 (default)	Unsafe	Unknown	No	No	Yes
fast=2	Very Unsafe	Unknown	No	No	Yes
except except-	*/** *	* *	* *	Yes No	* *

* These modes are unaffected. `–fp-model except[-]` only affects the precise FP exceptions mode.

** It is illegal to specify `–fp-model except` in an unsafe value safety mode.

FP Model - Example

Using **`-fp-model [precise|strict]`**:

- Disables reassociation
- Enforces standard conformance (left-to-right)
- May carry a significant performance penalty

```
#include <iostream>
#define N 100

int main()   {
  float a[N], b[N];
  float c = -1., tiny = 1.e-20F;

  for (int i=0; i<N; i++) a[i]=1.0;

  for (int i=0; i<N; i++)  {
    a[i] = a[i] + c + tiny;
    b[i] = 1/a[i];
  }

  std::cout << "a = " << a[0]
            << "   b = " << b[0]
            << "\n";
}
```

Disabling of reassociation also impacts vectorization (e.g. partial sums)!

Other FP Options I

- Linux*, OS X*: **`–[no-]ftz`**, Windows*: **`/Qftz[-]`**
  Flush denormal results to zero
- Linux*, OS X*: **`-[no-]prec-div`**, Windows*: **`/Qprec-div[-]`**
  Improves precision of floating point divides
- Linux*, OS X*: **`-[no-]prec-sqrt`**, Windows*: **`/Qprec-sqrt[-]`**
  Improves precision of square root calculations
- Linux*, OS X*: **`-fimf-precision=name`**, Windows*: **`/Qimf-precision:name`**
  `high`, **`medium`**, **`low`**: Controls accuracy of math library functions
- Linux*, OS X*: **`-fimf-arch-consistency=true`**, Windows*: **`/Qimf-arch-consistency:true`**
  Math library functions produce consistent results on different processor types of the same architecture

Other FP Options II

- Linux*, OS X*: **`-fpe0`**, Windows*: **`/fpe:0`**
  Unmask floating point exceptions (Fortran only) and disable generation of denormalized numbers

- Linux*, OS X*: **`-fp-trap=common`**, Windows*: **`/Qfp-trap:common`**
  Unmask common floating point exceptions (C/C++ only)

- Linux*, OS X*: **`-[no-]fast-transcendentals`**, Windows*: **`/Qfast-transcendentals[-]`**
  Enable/disable fast math functions

- …

Recommendation

- The **default FP model** is fast but has less precision/reproducibility (vectorization)
- The **strict FP model** has best precision/reproducibility but is slow (no vectorization; x87 legacy)
- For best trade-off between precision, reproducibility & performance use:
  Linux*, OS X*: **`-fp-model precise -fp-model source`**
  Windows*: **`/fp:precise /fp:source`**
  Approx. 12-15% slower performance for SPECCPU2006fp
- Don’t mix math libraries from different compiler versions!
- Using different processor types (of same architecture), specify:
  Linux*, OS X*: **`-fimf-arch-consistency=true`**
  Windows*: **`/Qimf-arch-consistency:true`**

More information:
http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler

How to Align Data (Fortran)

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

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

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

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

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

And tell the compiler…

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

- Asks compiler to vectorize, assuming all array data accessed in loop are aligned for targeted processor
  - May cause fault if data are not aligned

```
!dir$ assume_aligned array:n   [,array2:n2, …]
```

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

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

How to Align Data (C/C++)

Allocate memory on heap aligned to n byte boundary:

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

Alignment for variable declarations:

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

And tell the compiler…

```
#pragma vector aligned
```

- Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
- May cause fault if data are not aligned

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

- Compiler may assume array is aligned to n byte boundary

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

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:

```
#pragma omp simd [clause [,clause]…]
  for-loop
```

- 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`**, … must be power of 2: The compiler can assume a vectorization for a vector length of **`n1`**, **`n2`**, … 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`**, … etc. Therefore it is incremented by **`step1 * vector length`** for the vectorized loop.
- **`reduction(operator:v1, v2, …)`**
  Variables **`v1`**, **`v2`**, … etc. are reduction variables for operation **`operator`**
- **`collapse(n)`**: Combine nested loops – collapse them
- **`aligned(v1:base, v2:base, …)`**: Tell variables **`v1`**, **`v2`**, … are aligned; (default is architecture specific alignment)

Pragma SIMD Example

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

```
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:
  ```
  #pragma omp declare simd [clause [,clause]…]
    function definition or declaration
  ```

- 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!

- **Remember:**
  Both the function definition as well as the function declaration (header file) need to be specified like this!

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:

```
#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++)
```

```
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
```

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

FMAs

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 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.

Reproducibility: the bottom line (for Intel64)

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

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

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

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

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 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
```

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

Thank you!

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.

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