Consistency of Floating-Point Results using the Intel® Compiler
or
Why doesn't my application always give the same answer?

Dr. Martyn J. Corden
David Kreitzer

Software Services Group
Intel Corporation

Introduction

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 result of the exact calculation
        - Usually measured in fractional error, or sometimes "units in the last place" (ulp).
- 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 processor or operating system to another
- Performance
    - Produce an application that runs as fast as possible

These objectives usually conflict! However, good programming practices and judicious use of compiler options allow you to control the tradeoffs.

For example, it is sometimes useful to have a degree of reproducibility that goes beyond the inherent accuracy of a computation. Some software quality assurance tests may require close, or even bit-for-bit, agreement between results before and after software changes, even though the mathematical uncertainty in the result of the computation may be considerably larger. The right compiler options can deliver consistent, closely reproducible results while preserving good (though not optimal) performance.

Floating-Point Semantics

The Intel® Compiler implements a model for floating-point semantics based on the one introduced by Microsoft. [1] A compiler switch (/fp: for Windows*, -fp-model for Linux* or OS* X) lets you choose the floating-point semantics at a coarse granularity. It lets you choose the compiler rules for:

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

These map to the following arguments of the /fp: (-fp-model) switch[2]:

•	precise	allows value-safe optimizations only
•	source	specify the intermediate precision
	double	used for
	extended	floating-point expression evaluation
•	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”
•	consistent	best reproducibility from one processor type or
		set of build options to another (compiler version ≥17)
•	fast [=1]	allows value-unsafe optimizations
	(default)	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

This switch supersedes a variety of switches that were implemented in older Intel compilers, such as /Op and /flt-consistency (-mp and -flt-consistency).

[1] Microsoft* Visual C++* Floating-Point Optimization
http://msdn2.microsoft.com/en-us/library/aa289157(vs.71).aspx

[2] In general, the Windows form of a switch is given first, followed by the form for Linux and OS X in parentheses.

The recommendation for obtaining floating-point values that are compliant with ANSI / IEEE standards for C++ and Fortran is:

```
/fp:precise  /fp:source                   (Windows)
-fp-model precise -fp-model source        (Linux or OS X)
```

For C and C++, the float_control pragma may be used to achieve the same effect as the switches /fp:precise and /fp:fast (-fp-model precise and -fp-model fast):

```
#pragma float_control (precise, on)
#pragma float_control (precise, off)
   or
#pragma float_control (push)
#pragma float_control (precise, on)
#pragma float_control (pop)
```

Although such a pragma may be placed in front of an individual block of code, it applies to the entire containing function.

Value Safety

In SAFE mode, corresponding to the precise, strict or consistent arguments for /fp: (-fp-model), the compiler may not make any transformations that could affect the result. For example, the following is prohibited:

$$(x + y) + z \Rightarrow x + (y + z)$$

since general reassociation is not value safe. When the order of floating-point operations is changed, (reassociation), different intermediate results get rounded to the nearest floating-point representation, and this can lead to slight variations in the final result.

UNSAFE (fast) mode is the default. The variations implied by “unsafe” are usually very tiny; however, their impact on the final result of a longer calculation may be amplified if the algorithm involves cancellations (small differences of large numbers), as in the first example below. In such circumstances, the variations in the final result reflect the real uncertainty in the result due to the finite precision of the calculation.

VERY UNSAFE (fast=2) mode enables riskier transformations. For example, this might enable expansions that could overflow at the extreme limit of the allowed exponent range.

More Examples that are disabled by /fp:precise (-fp-model precise)

- reassociation e.g. (a + b) + c ⇨ a + (b + c)
- vectorization of reductions e.g. for(i=0; i<n; i++) sum += x[i];
  (special case of reassociation)
- zero folding e.g. X+0 ⇨ X, X*0 ⇨ 0
- multiply by reciprocal e.g. A/B ⇨ A*(1/B)
- approximate square root
- fast transcendental functions e.g. sin(a) or exp(a)
  (includes vectorization of loops containing transcendental functions)
- abrupt underflow (flush-to-zero)
- drop precision of RHS to that of LHS
- etc.

The zero folding examples above might not give the correct IEEE result for certain special values of X, such as infinity or NaN (Not a Number).
Note, however, that fused-multiply-add contractions[1] are still permitted unless they are explicitly disabled or /fp:strict (-fp-model strict) is specified. See the "Floating-Point Contractions" section.

More about Reassociation

Addition and multiplication are associative:

$$a + b + c = (a+b) + c = a + (b+c)$$
$$(a*b) * c = a * (b*c)$$

These transformed expressions are equivalent mathematically, but they are **not** equivalent in finite precision arithmetic. The same is true for other algebraic identities such as

$$a*b + a*c = a * (b+c)$$

Examples of higher level optimizing transformations that involve reassociation are loop interchange and the vectorization of reduction operations by the use of partial sums (see the section on Reductions below). The corresponding

[1] Certain Intel® microarchitectures support multiplication followed by an addition in a single instruction with a single rounding operation.

compiler options are available for both Intel® and non-Intel microprocessors but they may result in more optimizations for Intel microprocessors than for non-Intel microprocessors.

The ANSI C and C++ language standards do not permit reassociation by the compiler; even in the absence of parentheses, floating-point expressions are to be evaluated from left to right. Reassociation by the Intel compiler may be disabled in its entirety by the switch /fp:precise (-fp-model precise). This also disables other value-unsafe optimizations, and may have a significant impact on performance at higher optimization levels. Under the default setting of /fp:fast (-fp-model fast), the Intel compiler may reassociate expressions, even in the presence of parentheses.

The ANSI Fortran standard is less restrictive than the C standard: it requires the compiler to respect the order of evaluation specified by parentheses, but otherwise allows the compiler to reorder expressions as it sees fit. The Intel Fortran compiler has therefore implemented a corresponding switch, /assume:protect_parens (-assume protect_parens), that results in standard-conforming behavior for reassociation, with considerably less impact on performance than /fp:precise (-fp-model precise). This switch does not affect any value-unsafe optimizations other than reassociation. In the version 18 compiler, /Qprotect-parens (-fprotect-parens) is a synonym for /assume:protect_parens (-assume protect_parens).
Since the version 16 compiler, a similar option is available for the Intel® C/C++ compiler, /Qprotect-parens (-fprotect-parens). This requires the compiler to respect parentheses when determining the order of evaluation of expressions. The compiler may still reorder expressions or subexpressions not explicitly protected by parentheses, if allowed by the /fp (-fp-model) setting.

Example from a Fortran application

The application gave different results when built with optimization compared to without optimization, and the residuals increased by an order of magnitude. The root cause was traced to source expressions of the form:

```
A(I) + B + TOL
```

where TOL is very small and positive and A(I) and B may be large. With optimization, the compiler prefers to evaluate this as

```
A(I) + (B + TOL)
```

because the constant expression (B+TOL) can be evaluated a single time before entry to the loop over I. However, the intent of the code was to ensure that the expression remained positive definite in the case that A(I) ≈ -B. When TOL is added directly to B, its contribution is essentially rounded away due to the finite precision, and it no longer fulfills its role of keeping the expression positive-definite when A(I) and B cancel.

The simplest solution was to recompile the affected source files with the switch -fp-model precise, to disable reassociation and evaluate expressions in the order in which they are written. A more targeted solution, with less potential impact on performance, was to change the expression in the source code to

```
(A(I) + B) + TOL
```

to more clearly express the intent of the programmer, and to compile with the option -assume protect_parens.

Example from WRF[1] (Weather Research and Forecasting model)

Slightly different results were observed when the same application was run on different numbers of processors under MPI (Message Passing Interface).

This was because loop bounds, and hence data alignment, changed when the problem decomposition changed to match the different number of MPI processes. This in turn changed which loop iterations were in the vectorized loop kernel and which formed part of the loop prologue (“peel loop”) or epilogue (“remainder loop)”. Different generated code in the prologue or epilogue compared to the vectorized kernel can give slightly different results for the same data.

The solution was to compile with -fp-model precise. This causes the compiler to generate consistent code and math library calls for the peel loop, remainder loop and kernel. In some cases, this may prevent the loop from being vectorized.

Reductions

Parallel implementations of reduction loops (such as dot products) make use of partial sums, which implies reassociation. They are therefore not value-safe. The following is a schematic example of serial and parallel implementations of a floating-point reduction loop:

[1] See http://www.wrf-model.org

```
float Sum(const float A[ ], int n )
{

   float sum=0;
   for (int i=0; i<n; i++)
      sum = sum + A[i];







   return sum;
}
```

```
float Sum( const float A[ ], int n )
{
   int i,  n4 = n-n%4;
   float sum=0,sum1=0,sum2=0,sum3=0;
   for (i=0; i<n4; i+=4) {
        sum   = sum    + A[i];
        sum1 = sum1 + A[i+1];
        sum2 = sum2 + A[i+2];
        sum3 = sum3 + A[i+3];
   }
   sum = sum + sum1 + sum2 + sum3;
   for (i=n4; i<n; i++) sum = sum + A[i];
        return sum;
}
```

In the second implementation, the four partial sums may be computed in parallel, either by using SIMD instructions (eg as generated by the compiler's automatic vectorizer), or by a separate thread for each sum (e.g. as generated by automatic parallelization). This can result in a large increase in performance; however, the changed order in which the elements of A are added to give the final sum results in different rounding errors, and thus may yield a slightly different final result.
Because of this, the vectorization or automatic parallelization of reductions is disabled by /fp:precise (-fp-model precise), except where vectorization is explicitly mandated by an OpenMP* SIMD pragma or directive with a REDUCTION clause (see section on vectorization using OpenMP SIMD pragmas or directives).

Parallel reductions in OpenMP are mandated by the OpenMP directive, and cannot be disabled by /fp:precise (-fp-model precise). Generally speaking, they are value-unsafe, and remain the responsibility of the programmer. Likewise, MPI* reductions involving calls to an MPI library are beyond the control of the compiler, and might not be value-safe. Changes in the number of threads, in the scheduling method or in the number of MPI processes are likely to cause small variations in results. In some cases, the order of operations may change between consecutive executions of the same binary.

The OpenMP standard does not specify the order in which partial sums should be combined, which may therefore be decided at runtime and vary from run to run. Versions 13 and later of the Intel Compiler provide a method to ensure consistent, reproducible results from OpenMP reductions for repeated executions of the same binary, for a fixed number of threads and static scheduling only. The following environment variable should be set:

```
KMP_DETERMINISTIC_REDUCTION=yes        (or =on  or =true or =1 )
```

This also tends to increase the accuracy of large reductions. For large numbers of threads, KMP_DETERMINISTIC_REDUCTION=yes is the default; for small numbers of threads, it is not, as there may be a slight impact on performance.

The Intel® Math Kernel Library, (Intel® MKL), and Intel® Threading Building Blocks, (Intel® TBB), in Intel® Parallel Studio XE 2013 and later contain similar functionality to provide reproducible results for repeated parallel execution of the same binary. For more detail, please consult the documentation for Conditional Numerical Reproducibility (Intel MKL) and parallel_deterministic_reduce() (Intel TBB).

Compiler options that enable vectorization and OpenMP are available for both Intel® and non-Intel microprocessors but they may result in more optimizations for Intel microprocessors than for non-Intel microprocessors.

Second Example from WRF

Slightly different results were observed when re-running the same (non-threaded) binary on the same data on the same processor.

This was caused by variations in the starting address and alignment of the global stack, resulting from events external to the program. The resulting change in local stack alignment led to changes in which loop iterations were assigned to the loop prologue or epilogue, and which to the vectorized loop kernel. This in turn led to changes in the order of operations for vectorized reductions (i.e., reassociation).

The solution was to build with -fp-model precise, which disabled the vectorization of reductions.

Starting with version 11 of the Intel compiler, the starting address of the global stack is aligned to a cache line boundary. This avoids the run-to-run variations described above, even when building with /fp:fast (-fp-model fast), unless run-to-run variations in stack alignment occur due to events internal to the application. (This might occur if a variable length string is allocated on the stack to contain the current date and time, for example). Dynamic variations in heap alignment can lead to variations in floating-point results in a similar manner. Such variations in alignment typically arise from memory allocations that depend on the external environment. They can be prevented from causing variations in floating-point results by building with /fp:precise (-fp-model precise), or by explicit alignment of data arrays. Starting from the version 15 compiler, such run-to-run variations can also be prevented by compiling with /Qopt-dynamic-align- (-qno-opt-dynamic-align), which may have much less impact on performance than /fp:precise (-fp-model precise).

Vectorization using OpenMP* SIMD Pragmas or Directives

In some circumstances, OpenMP pragmas or directives that require vectorization of a loop can be in conflict with requirements of the Floating-point model specified by a /fp (-fp-model) switch or by /Qfp-speculation:safe (-fp-speculation=safe). By default, when OpenMP SIMD pragmas are enabled, they take precedence and in some cases lead to floating-point results that differ from results obtained without the pragma.
To avoid this, certain OpenMP SIMD constructs must be avoided.

Floating-Point Reductions

As an example, take a variant of the simple reduction loop considered previously.

```
        for (int i=0; i<n; i++)    sum = sum + A[i]*A[i]*A[i];
```

When compiled with -fp-model precise or -fp-model consistent, the loop is not auto-vectorized with the following message:

```
remark #15331: loop was not vectorized: precise FP model implied by the command line or a directive prevents vectorization.
```

However, if we place an OpenMP SIMD pragma before the loop,

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

We then see

```
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
```

As discussed in the section on “Reductions”, the vectorization of the reduction loop changes the order of operations and yields results that may differ from lower optimization levels such as -O1, at which the loop would not be vectorized. The SIMD pragma can be prevented from overriding the floating-point model and consistent results can be preserved using the compiler option /Qsimd-honor-fp-model (-qsimd-honor-fp-model). The loop is no longer vectorized with the message:

```
   remark #15331: simd loop was not vectorized: precise FP model implied by the command line or a directive prevents vectorization. Consider using fast FP model
   remark #15552: loop was not vectorized with "simd"
```

For loops that contain a lot of computation in addition to the reduction, it may be worthwhile to compute the reduction serially, so that results are reproducible, while vectorizing the remainder of the loop to improve performance. This can be achieved by adding the compiler option /Qsimd-serialize-fp-reduction (-qsimd-serialize-fp-reduction), which allows statements other than the reduction statement to be vectorized.

For reproducible results with -qsimd-serialize-fp-reduction, there must be only a single lexical statement in the loop that updates the reduction variable; it must be executed only once per loop iteration; and it should contain no function calls. For better performance, it is recommended to minimize the amount of computation in the reduction statement itself by keeping it as simple as possible and using temporary variables if necessary.
E.g., if the example above is rewritten as

```
        double temp = A[i]*A[i]*A[i];
        sum += temp;
```

then the multiplications are carried out in SIMD mode but the additions are performed serially. The remarks in the optimization report are:

```
   remark #15531: Block of statements was serialized due to user request
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
```

Early Exit Clause

A loop with multiple exits, such as a search loop, can often be vectorized using a pragma or directive such as:

```
    #pragma omp simd early_exit
```

However, the compiler may speculatively execute code that comes lexically before a break statement. This could have consequences such as exceptions that would not occur in the absence of speculation. If this could be a problem, avoid using the early_exit clause.

Speculation occurring in loops that are auto-vectorized without an OpenMP SIMD pragma can be disabled by the compiler switch -fp-speculation=safe (see the section on the Floating-Point Environment). However, this switch may not disable speculation resulting from an OpenMP early_exit pragma.

Ordered Clause

Multiple “ordered” constructs within one OpenMP SIMD loop could cause operations to occur in a different order to the one that would obtain for a loop without SIMD pragmas. To ensure that results remain reproducible, do not use more than one ordered construct (e.g., #pragma omp ordered simd) within a single loop.

Overlap Modifiers

The overlap modifier enables generation of more efficient code for architectures that natively support conflict detection instructions such as Intel® AVX-512CD. This optimization may cause variations in the order of operations and hence variations in floating-point results. To maintain reproducibility of results, do not use overlap modifiers, e.g.

```
    #pragma omp ordered simd overlap(b[i])
```

in conjunction with floating-point data.

Private and Lastprivate Clauses

Private and lastprivate clauses define objects whose values live for one iteration only. The OpenMP standard says that an uninitialized object is created for each SIMD lane. If such an object is accessed before it has been initialized, results are unspecified and non-deterministic. Also, the number and order of constructor or destructor calls is unspecified.

To maintain reproducibility of results, initialize private or lastprivate objects inside the loop and do not rely on the number or order of constructor or destructor calls for these.

Abrupt Underflow or Flush-To-Zero (FTZ)

Denormalized numbers[1] (denormals) extend slightly the allowed range of floating-point exponents, but computations involving them take substantially longer than those that involve only normal numbers. By default, when the result of a floating-point calculation would have been a denormal, it is instead set to

[1] A short discussion of denormal numbers may be found in the Floating-Point Operation section of the Intel Compiler User and Reference Guides.

zero in hardware. When /fp:precise (-fp-model precise) is specified, denormal results are preserved for value safety.

The /fp: (-fp-model) settings may be overridden for the entire program by compiling the main function or routine with the switch /Qftz (-ftz) or /Qftz- (-no-ftz), which sets or unsets the hardware flush-to-zero mode in the floating-point control register[1]. The default setting for /fp:fast (-fp-model fast) is /Qftz (-ftz) for optimization levels of -O1 and above.

There is no flush-to-zero hardware for x87 arithmetic, for which the /Qftz (-ftz) switch has no effect. x87 arithmetic instructions are usually generated only in special circumstances, such as when compiling for older IA-32 processors without Intel SSE2 support using the option /arch:ia32 (-mia32).

Floating-Point Expression Evaluation

Example: a = (b + c) + d

There are four possibilities for rounding of the intermediate result (b+c), corresponding to values of FLT_EVAL_METHOD in C99:

Evaluation Method	**/fp: (-fp-model)**	**Language**	**FLT_EVAL_METHOD**
Indeterminate	fast	C/C++/Fortran	-1
Use source precision	source	C/C++/Fortran	0
Use double precision	double	C/C++	1
Use long double precision	extended	C/C++	2

If /fp:precise (-fp-model precise) is specified but the evaluation method is not, the evaluation method defaults to source precision on Intel64 architecture. For C/C++ on IA-32 architecture, the evaluation method defaults to double on Windows and to extended on Linux[2]. For Fortran, source is the only supported evaluation method under /fp:precise (-fp-model precise). If an evaluation method of source, double or extended is specified but no value safety option is given, the latter defaults to /fp:precise (-fp-model precise).

[1] The switch /Qftz (-ftz) allows denormals results to be flushed to zero. It does not guarantee that they will always be flushed to zero. It also sets denormal inputs to zero (DAZ).

[2] The switch -mia32 is not supported on Mac OS* X, where all Intel processors support instructions up to Intel SSE3. The evaluation method therefore defaults to source precision with -fp-model precise.

The method of expression evaluation can impact performance, accuracy, reproducibility and portability! In particular, selection of an evaluation method that implies repeated conversions between representations of different precision can significantly impact performance.

The Floating-Point Unit (FPU) Environment

The floating-point environment[1] consists of the floating-point control word settings and status flags. The control word settings govern:

- the FP rounding mode (nearest, toward +∞, toward -∞, toward 0)
- FP exception masks for inexact, underflow, overflow, divide by zero, denormals and invalid exceptions
- Flush-to-zero (FTZ), Denormals-are-zero (DAZ)
- For x87[2] only: precision control (single, double, extended)
    - Changing this may have unintended consequences!

There is a status flag corresponding to each exception mask.

Programmer access to the FPU environment is disallowed by default.

- the compiler assumes the default FPU environment:
    - round-to-nearest
    - all FP exceptions are masked
    - Flush-to-zero (FTZ) and Denormals-as-zero (DAZ) are disabled
- the compiler assumes the program will not read FP status flags

If the user might explicitly change the default FPU environment, e.g. by a call to the runtime library that modifies the FP control word, the compiler must be informed by setting the FPU environment access mode. The access mode may only be enabled in value-safe modes, by either

- /fp:strict (-fp-model strict) or
- #pragma STDC FENV_ACCESS ON (C/C++ only)

In this case, the compiler treats the FPU control settings as unknown. It will preserve floating-point status flags and disable certain optimizations such as the evaluation of constant expressions at compile time, speculation of floating-point operations and others[3]. Changing the default floating-point environment

[1] For more detail, see the Intel Compiler User and Reference Guides, under Floating-point Operations/Understanding Floating-point Operations/Floating-point Environment.

[2] There is a separate control word for x87 floating-point arithmetic. The x87 FP control word should not normally be of concern unless the /arch:IA32 (-mia32) option for the support of older processors is specified.

[3] Other optimizations that are disabled:
Partial redundancy elimination
Common subexpression elimination
Dead code elimination

without informing the compiler may lead to unpredictable results, e.g. for math library functions if changes are made to the rounding mode.

Example of changing the FPU environment:

```
#include <fenv.h>
double  x[20][20],  zero = 0.;
  feenableexcept(FE_DIVBYZERO);
  for( int i = 0; i < 20; i++ )
    for( int j = 0; j < 20; j++)
      x[i][j] = zero ? (1./zero) : zero;
    .....
```

A floating-point exception may occur, despite the explicit protection, because the calculation of (1./zero) gets hoisted out of the loop by the optimizer, so that it is only evaluated once, but the branch implied by "?" remains in the loop. The compiler assumes that this is safe, because divide-by-zero exceptions are masked in the default FPU environment. If the default environment is modified, as here by the call to feenableexcept(), the compiler should be informed, either by compiling with the option /fp:strict (-fp-model strict),
or by use of the pragma

```
        #pragma STDC FENV_ACCESS ON       (C/C++ only).
```

The optimization leading to the premature exception may also be disabled more directly with the option /Qfp-speculation:safe (-fp-speculation safe). This avoids some of the other consequences of /fp:strict (-fp-model strict), such as suppression of fused multiply-add instructions and non- vectorization of loops containing reductions or math functions. The following example occurs frequently:

```
double  *a, *b;
   for( int i = 0; i < 100; i++ )
       if (a[i] != 0.)  b[i] = b[i] / a[i]
```

At default optimization, the compiler will vectorize this loop using Intel® SSE instructions. To do this, it uses packed SIMD instructions to speculatively evaluate b[i]/a[i] for all values of i, but then stores the result back to b[i] only for those values of i for which the mask is true. This is safe, because the divide-by-zero exception is masked in the default FPU environment. If it is explicitly

Conditional transform, e.g.   if (c) x = y; else x = z; ➔ x = (c) ? y : z;

unmasked, e.g. by a call to feenableexcept() or by a command line switch such as /Qfp-trap:common (-fp-trap =common), the exception will be trapped and the program will terminate. This can be avoided by compiling with /Qfp-speculation:safe (-fp-speculation safe), which will disable vectorization where there is a risk that speculation might lead to an exception. Note that more recent instruction sets, such as Intel® Advanced Vector Extensions 512 (Intel® AVX-512), include SIMD instructions that are masked in hardware. These may allow some loops similar to the above to be vectorized without the need for speculation.

Precise Floating-Point Exceptions

By default, (precise exceptions disabled), code may be reordered by the compiler during optimization, and so floating-point exceptions might not occur at the same time and place as they would if the code were executed exactly as written in the source. This effect is particularly important for x87 arithmetic where exceptions are not signaled as promptly as for Intel SSE or Intel AVX.

Precise FP exceptions may be enabled by one of:

- /fp:strict (-fp-model strict)
- /fp:except (-fp-model except)
- #pragma float_control(except, on) (C and C++ only)

When enabled, the compiler must account for the possibility that any floating-point operation might throw an exception. Optimizations such as speculation of FP operations are disabled, as these might result in exceptions coming from a branch that would not otherwise be executed. This may prevent the vectorization of certain loops containing “if” statements, for example. The compiler inserts fwait after other x87 instructions, to ensure that any FP exception is synchronized with the instruction causing it. Precise FP exceptions may only be enabled in value-safe mode, i.e. with /fp:precise (-fp-model precise) or #pragma float_control(precise, on). Value-safety is already implied by /fp:strict (-fp-model strict).

Note that enabling precise FP exceptions does not unmask FP exceptions. That must be done separately, e.g. with a function call, or (for Fortran) with the command line switch /fpe:0 (-fpe0) or /fpe-all:0 (-fpe-all0), or (for C or C++) with a command line switch such as /Qfp-trap:common (-fp-trap=common) or /Qfp-trap-all:common (-fp-trap-all=common).

Floating-Point Contractions

This refers primarily to the generation of fused multiply-add (FMA) instructions, such as found in the Intel® AVX2 and Intel® AVX-512 instruction sets. FMA generation is enabled by default. The compiler may generate a single FMA instruction for a combined multiply and add operation,

e.g.   a = b*c + d.

This leads to faster, slightly more accurate calculations, but results may differ in the last bit from separate multiply and add instructions, if all terms are positive, or by much more, if there is a cancellation.

Floating-point contractions are enabled by default at optimization levels /O1 (-O1) and above when targeting processors that support the FMA instructions, for example when compiling with switches such as /QxCORE-AVX2 (-xcore-avx2). They are a common source of differences in floating-point results compared to older processors. Generation of FMA instructions may be disabled at the source file or function level by one of the following:

- /fp:strict (-fp-model strict)
- #pragma fp_contract(off)   (C/C++)
- !DIR$ NOFMA   (Fortran)
- /Qfma-   (-no-fma)   (this overrides the /fp or -fp-model setting)

When disabled, the compiler must generate separate multiply and add instructions, with rounding of the intermediate result. Note that generation of FMA instructions is **not** disabled by /fp:precise (-fp-model precise).

The Intel C/C++ compiler supports SIMD intrinsics for FMA generation, such as _mm256_fmadd_pd() etc., for processors that support FMA instructions. However, intrinsics may still be subject to further compiler optimization. Writing an addition and a multiplication as separate assignments on consecutive source lines does not prevent FMA instructions from being generated. If needed for debugging, a “memory fence” intrinsic may be used to prevent FMA generation by forcing an intermediate result to be stored to memory, for example:

```
t = a*b;
_mm_mfence();
result = t + c;
```

The fma() and fmaf() intrinsics from math.h should always give a result with a single rounding, even on processors with no FMA instruction, though at some cost in performance.

FMA instructions may sometimes break the symmetry of an expression.
Consider:

```
        c = a;    d = -b;
        result = a*b + c*d;
```

This will normally result in zero in the absence of FMA instructions.
If FMAs are supported, the compiler may convert this 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.

Typical Performance Impact of /fp:precise /fp:source (-fp-model precise -fp-model source)

The options /fp:precise /fp:source /Qftz (-fp-model precise -fp-model source -ftz) are recommended to improve floating-point reproducibility while limiting performance impact, for typical applications where the preservation of denormalized numbers is not important. The switch /fp:precise (-fp-model precise) disables certain optimizations, and therefore tends to reduce application performance. The performance impact may vary significantly from one application to another. It tends to be greatest for applications with many high level loop optimizations, since these often involve the reordering of floating-point operations. The impact is illustrated by performance estimates for the SPECCPU2017 speed Floating Point benchmark suite. With build options including -O3 -xcore-avx2 -ipo, the geomean of the performance reduction due to -fp-model precise -fp-model source was about 9%. The geomean of the performance reduction due to -fp-model consistent -fp-model source was about 12%. This was using the Intel Compiler 18.0 on an Intel Xeon® E5-2680 v4 system with dual, 14-core processors at 2.40 GHz, 251GB memory and 35MB cache running Linux*. The option -O3 is available for both Intel® and non-Intel microprocessors but it may result in more optimizations for Intel microprocessors than for non-Intel microprocessors. The option -xcore-avx2 is available only for Intel microprocessors.

Additional Remarks

The options /fp:precise /fp:source (-fp-model precise -fp-model source) should also be used for debug builds at /Od (-O0). In particular, the evaluation method may not default to “source” at /Od (-O0).

Although the floating-point model described in this paper is also applicable to Intel® MIC™ Architecture, there are some slight differences in implementation for the Intel® Xeon Phi™ coprocessor x100 family. These are described in the paper "Differences in floating-point arithmetic between Intel® Xeon® processors and the Intel® Xeon Phi™ coprocessor", see the "Further Information" section below. There are no differences in the implementation of the floating-point model between the Intel® Xeon Phi™ x200 processor family and the Intel® Xeon® processor family.

Math Library Functions

As yet, no C, C++ or Fortran standard specifies the accuracy of mathematical functions[1] such as log() or sin(), or how the results should be rounded. Different implementations of these functions may not have the same accuracy or be rounded in the same way.

The Intel compiler may implement math functions in the following ways:

- By standard calls to the optimized Intel math library libm (on Windows) or libimf (on Linux or OS X). These calls are mostly compatible with math functions in the Microsoft C runtime library libc (Windows) or the GNU library libm (Linux or OS X).
- By generating inline code that can be optimized in later stages of the compilation
- By architecture-specific calling sequences (e.g. by passing arguments via SIMD registers on IA-32 processors with support for Intel SSE2)
- By vector calls to the short vector math library (libsvml) for loops that can be vectorized
- By scalar calls to the short vector math library (libsvml)

Calls may be limited to the first of these methods by the switch /fp:precise (-fp-model precise) or by the more specific switches /Qfast-transcendentals- (-no-fast-transcendentals). This makes the calling sequence generated by the compiler consistent between different optimization levels or different compiler versions. However, it does not ensure consistent behavior of the library function itself. The value returned by a math library function may vary:

- Between one compiler release and another, due to algorithmic and optimization improvements

[1] With the exception of division and square root functions.

- Between one run-time processor and another. The math libraries contain function implementations that are optimized differently for different processors. In the Intel® Compiler version 17 and earlier, the code automatically detects what type of processor it is running on, and selects the implementation accordingly. For example, a function involving complex arithmetic might have implementations both with and without Intel SSE3 instructions. The implementation that used Intel SSE3 instructions would be invoked only on a processor that was known to support these. In the version 18 compiler, for processor targets supporting at least Intel® AVX, the function implementation is determined at compile time by the processor targeting switch (/Qx, /Qax or /arch on Windows; -x, -ax or -m on Linux) for calls resolved by the Short Vector Math Library (libsvml). It does not depend on the processor on which the application is executed. Processor targeting at run-time, as in the 17.0 and earlier compilers, can be enabled by the new option /Qimf-force-dynamic-target (-fimf-force-dynamic-target).
- Math function calls potentially yield different results depending on whether they are resolved from the Short Vector Math Library libsvml or Intel's optimized scalar libraries libm (Windows) or libimf (Linux). To ensure that equivalent calls always yield the same results, resolution can be restricted to libm (libimf) by /fp:precise (-fp-model precise) or /Qfast-transcendentals- (-no-fast-transcendentals). This typically prevents vectorization of loops containing such calls, which may result in significant loss of performance. In the version 18 compiler, the switch /Qimf-use-svml (-fimf-use-svml) will try instead to resolve all math function calls only from libsvml, whenever SVML supports that particular function. This allows loops containing such calls to be vectorized while preserving identical results for scalar and vector calls, including calls from within peel or remainder loops. The Intel Compiler Developer Guide and Reference contains additional discussion of the side effects of this option on accuracy, errno generation and floating-point exceptions.

The variations in the results of math functions discussed above are small. The expected accuracy is maintained both for different compiler releases and for implementations optimized for different processors and is governed by the /Qimf-max-error or /Qimf-precision:high | medium | low
(-fimf-max-error or -fimf-precision=high | medium | low) switches.

There is no direct way to enforce bit-for-bit consistency between math libraries coming from different compiler releases. It may sometimes be possible to use the runtime library from the higher compiler version in conjunction with both compilers when checking for consistency of compiler generated code.

The switch /Qimf-arch-consistency:true (-fimf-arch-consistency=true) may be used to ensure bit-wise consistency between results returned by math library functions on different processor types of the same architecture, including between results on Intel processors and on compatible, non-Intel processors. This switch does not ensure bit-wise consistency between different architectures, such as between IA-32 and Intel 64. This switch may result in reduced performance, since it results in calls to less optimized functions that can execute on a wide range of processors.

Because the version 18 compiler defaults to compile-time targeting of SVML math functions for Intel AVX and higher instruction sets, (see above), another approach becomes possible. The options /Qxcore-avx2 /Qimf-use-svml (-xcore-avx2 -fimf-use-svml) would give consistent math function results on different, Intel processors supporting at least Intel AVX2. This might result in better performance than /Qimf-arch-consistency:true (-fimf-arch-consistency=true), since it would allow the use of more optimized math functions, for example making use of FMA instructions.

Adoption of a formal standard with specified rounding for the results of math functions would encourage further improvements in floating-point consistency, including between different architectures, but would likely come at an additional cost in performance.

The Intel compiler contains additional options to control the accuracy of results returned by math functions, such as /Qimf-precision (-fimf-precision) and /Qimf-max-error (-fimf-max-error), that may have an indirect impact on reproducibility. For more details of these options, see the Intel Compiler User and Reference Guides.

Vectorization of loops containing math functions such as log() or sin() is normally disabled by /fp:precise (-fp-model precise), since vectorization would result in a function call to a different math library, libsvml, that returns different, slightly less accurate results than the default optimized Intel math library (libm or libimf, see above). This could also lead to differences in results between vectorized and non-vectorized loops, and consequently to differences between different optimization levels.
The version 18 compiler implements a new option, /Qimf-use-svml (Windows) or -fimf-use-svml (Linux or OS X). This causes the Short Vector Math Library

(libsvml) to be used for all math functions, including for scalar calls that would otherwise have been resolved from the default Intel math library libm or libimf. This has the advantage that math function calls yield the same result, whether or not they are part of a vectorized loop. Consequently, vectorization of loops containing math functions is not disabled when compiling with /fp:precise /Qimf-use-svml (-fp-model precise -fimf-use-svml). The disadvantage is that the accuracy of scalar math function calls of better than 1 ulp, corresponding to the option /Qimf-precision:high (-fimf-precision=high), is very slightly worse than if they were resolved from the default optimized Intel math library libm or libimf.
The Short Vector Math Library is optimized for throughput, unlike libm and libimf which are optimized for latency. libsvml only supports the default rounding mode of round-to-nearest even, does not set error codes in the integer global variable errno and may sometimes raise exceptions that would not be seen with libm or libimf. For some functions, this may result in slightly better performance for scalar calls than for the default library; for other functions, performance may be worse.

The switch /Qimf-use-svml (-fimf-use-svml) introduced in the version 18.0 compiler is designed to be value-safe and should therefore be preferred to /Qfast-transcendentals (-fast-transcendentals) for re-enabling vectorization of transcendental math functions in the presence of /fp:precise (-fp-model precise).

/fp:precise (-fp-model precise) also implies the switches /Qprec-sqrt (-prec-sqrt) and /Qprec-div (-prec-div) that ensure the generation of consistent, correctly rounded results for square roots and for division. In some cases, this may disable vectorization of loops containing division or square root functions. The switches /Qprec-sqrt- (-no-prec-sqrt) and /Qprec-div- (-no-prec-div) may be used to re-enable vectorization. The accuracy of the generated code sequences may be also be controlled using the switch /Qimf-precision (-fimf-precision).

Many math library functions are more highly optimized for Intel microprocessors than for other microprocessors. While the math libraries in the Intel® Compiler offer optimizations for both Intel and compatible, non-Intel microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Bottom Line

Compiler options let you control the tradeoffs between accuracy, reproducibility and performance. Use /fp:precise /fp:source (Windows) or -fp-model precise -fp-model source (Linux or OS X) to improve the consistency and reproducibility of floating-point results while limiting the impact on performance[1]. If reproducibility between different processor types of the same architecture is important, use also /Qimf-arch-consistency:true (Windows) or -fimf-arch-consistency=true (Linux or OS X). For best reproducibility between different processors, at least one of which supports FMA instructions, use also /Qfma- (Windows) or -no-fma (Linux or OS X).

In the version 17 and later compilers, best reproducibility may be obtained with the single switch /fp:consistent (Windows) or -fp-model consistent (Linux or OS X), which sets all of the above options.

With the version 18 compiler, for applications with vectorizable loops containing math functions, it may be possible to improve performance whilst maintaining best reproducibility by adding /Qimf-use-svml (-fimf-use-svml).

Further Information

- Microsoft Visual C++* Floating-Point Optimization http://msdn2.microsoft.com/en-us/library/aa289157(vs.71).aspx
- Intel® MKL Conditional Numerical Reproducibility: https://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr
- Differences in Floating-Point Arithmetic Between Intel® Xeon® Processors and the Intel® Xeon Phi™ Coprocessor: https://software.intel.com/en-us/articles/differences-in-floating-point-arithmetic-between-intel-xeon-processors-and-the-intel-xeon
- The Intel® C++ and Fortran Compiler User and Reference Guides, "Floating-Point Operations" section.
- Goldberg, David: "What Every Computer Scientist Should Know About Floating-Point Arithmetic" *Computing Surveys*, March 1991, pg. 203

[1] /fp:source implies also /fp:precise

Optimization Notice

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

Notice revision #20110804

Intel, Xeon, Xeon Phi and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel http://www.intel.com/performance/resources/limits.htm

*Other names and brands may be claimed as the property of others. The linked sites are not under the control of Intel and Intel is not responsible for the content of any linked site or any link contained in a linked site. Intel reserves the right to terminate any link or linking program at any time. Intel does not endorse companies or products to which it links and reserves the right to note as such on its web pages. If you decide to access any of the third party sites linked to this Site, you do this entirely at your own risk. INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Appendix

Quick summary of main floating-point switches:

Primary Switch	Description
/fp:*keyword* **-fp-model *keyword***	***fast**[=**1**|2], precise, except, strict, consistent, source* *[double, extended - C/C++ only]* *Controls floating-point semantics*
Other Switches	
/Qftz[-] -[no-]ftz	*Flushes denormal results to zero*
/Qimf-use-svml -fimf-use-svml	*Use the Short Vector Math Library to resolve all (including scalar) calls to math library functions*
/Qprec-div[-] -[no-]prec-div	*Improves precision of floating-point divides*
/Qprec-sqrt[-] -[no-]prec-sqrt	*Improves precision of square root calculations*
/Qfp-speculation *keyword* -fp-speculation *keyword*	***fast**, safe, strict, off* *floating-point speculation control*
/fpe:0 -fpe0	*Unmask floating-point exceptions (Fortran only) and disable generation of denormalized numbers*
/Qfp-trap:*common* -fp-trap=*common*	*Unmask common floating-point exceptions (C/C++ only)*
/Qimf-arch-consistency:*true* -fimf-arch-consistency=*true*	*Math library functions produce consistent results on different processor types of the same architecture*
/Qfma[-] -[no-]fma	*Enable[Disable] use of fused multiply-add (FMA) instructions*
/Qopt-dynamic-align[-] -q[no-]opt-dynamic-align	*Enable[Disable] dynamic data alignment optimizations, that could possibly cause slight run-to-run variations in floating-point results*
/Qprotect-parens[-] -f[no-]protect-parens	*[Don't] Require the compiler to respect parentheses for the order of evaluation of expressions.*
/Qimf-precision:*name* -fimf-precision=*name*	*high, medium, low* *Controls accuracy of math library functions*
/Qimf-force-dynamic-target -fimf-force-dynamic-target	*Enables run-time dispatch of math functions. Code path depends on processor on which executed.*
/Qprec -mp1	*More consistent comparisons & transcendentals*
/Qfast-transcendentals[-] -[no-]fast-transcendentals	*Enable[Disable] optimization of math functions that might not be value-safe*

/Qsimd-honor-fp-model[-] -q[no-]simd-honor-fp-model	*Tells the compiler to obey the selected floating-point model when vectorizing SIMD loops.*
/Qsimd-serialize-fp-reduction[-] -q[no-]simd-serialize-fp-reduction	*Tells the compiler to serialize floating-point reductions when vectorizing SIMD loops.*