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 Solutions 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 Mac 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 used for
  double — floating point expression evaluation
  extended
- 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] — 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, /QIPF-fltacc, /flt-consistency (-mp, -IPF-fltacc, -flt-consistency).

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 Mac OS X)

[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 Mac OS X in parentheses.

Value Safety

In SAFE (precise) mode, 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)
- zero folding e.g. X+0 ⇨ X, X*0 ⇨ 0
- multiply by reciprocal e.g. A/B ⇨ A*(1/B)
- approximate square root
- abrupt underflow (flush-to-zero)
- drop precision of RHS to that of LHS
- etc

Note, however, that fused-multiply-add contractions[1] are still permitted unless they are explicitly disabled or /fp:strict (-fp-model strict) is specified.

More about Reassociation

Addition and multiplication are associative:

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

[1] Intel® Itanium®-based processors support multiplication followed by an addition in a single instruction.

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 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.

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. It is not available for the C/C++ compiler.

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 or epilogue. 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 loop prologue, epilogue and kernel. 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:

```
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 )
{
  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<n; i++) sum = sum + A[i];
  return sum;
  }
```

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

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).

Parallel reductions in OpenMP* are mandated by the OpenMP directive, and can not be disabled by /fp:precise (-fp-model precise). 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. Reducer hyper-objects in Intel® Cilk™ Plus are also not value-safe, the work-stealing model means that the order of operations could vary. Changes in the number of threads 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.

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-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).[1]

Abrupt Underflow or Flush-To-Zero (FTZ)

Denormalized numbers[2] 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 on IA-32 and Intel 64 architectures, when the result of a floating-point calculation using SSE instructions would have been a denormals number, it is instead set to zero in hardware. When /fp:precise (-fp-model precise) is specified, denormals 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[3]. The default setting for /fp:fast (-fp-model fast) on IA-32 and Intel 64 architectures is –ftz for optimization levels of –O1 and above. The default setting for the IA-64 architecture is –ftz only at –O3. The –ftz switch has no effect on x87 arithmetic, which has no flush-to-zero hardware. For the 11.1and later compilers, x87 arithmetic instructions are usually generated only when compiling for older IA-32 processors without 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

[1] Dynamic variations in heap alignment can lead to variations in floating-point results in a similar manner. These typically arise from memory allocations that depend on the external environment, and can also be avoided by building with /fp:precise (-fp-model precise).

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

[3] The switch /Qftz (-ftz) allows denormals results to be flushed to zero. It does not guarantee that they will always be flushed to zero.

If /fp:precise (-fp-model precise) is specified but the evaluation method is not, the evaluation method defaults to source precision, except in the special case of X87 code generation by the C/C++ compiler, for example when explicitly targeting an older IA-32 processor[1] that does not support SSE2, using the switch /arch:IA32 (-mia32). In this special case, the evaluation method defaults to double on Windows and to extended on Linux[2].
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).
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[3] 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[4] 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

---

[1] Such as an Intel Pentium® III processor.
[2] The switch –mia32 is not supported on Mac OS* X, where all Intel processors support instructions up to SSE3. The evaluation method therefore defaults to source precision with –fp-model precise.
[3] For more detail, see the Intel Compiler User and Reference Guides, under Floating-point Operations/Understanding Floating-point Operations/Floating-point Environment.
[4] There are two separate control words for SSE and x87 floating-point arithmetic. From the 11.1 compiler onwards, 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.

- /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[1].

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 SSE.

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 FP operation might throw an exception. Optimizations such as speculation of floating-point 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 floating-point 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++ in the 12.0 or later compilers) with a command line switch such as /Qfp-trap:common (-fp-trap=common) or /Qfp-trap-all:common (-fp-trap-all=common).

[1] Other optimizations that are disabled:
Partial redundancy elimination
Common subexpression elimination
Dead code elimination
Conditional transform, e.g. if (c) x = y; else x = z; ➔ x = (c) ? y : z;

Example of precise exceptions:

```
double  x,  zero = 0.;
  feenableexcept(FE_DIVBYZERO);
  for( int i = 0; i < 20; i++ )
    for( int j = 0; j < 20; j++)
      x = zero ? (1./zero) : zero;
    .....
```

A floating point exception occurred, 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 optimization leading to the premature exception may be disabled in one of the following ways:

- icc –fp-model precise –fp-model except (or icc –fp-model strict)
  This disables all optimizations that could affect FP exception semantics.
- icc –fp-speculation safe
  disables just speculation where this could cause an exception.
- #pragma float_control(except on|off) around the affected code block.

Floating Point Contractions

This refers primarily to the generation of fused multiply-add (FMA) instructions on the IA-64 architecture, which 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.

Floating point contractions may be disabled by one of the following:

- /fp:strict (-fp-model strict)
- #pragma float_control(fma,off)
- /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.

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 SPECCPU2006fp benchmark suite. When building with –O2 or –O3, the geomean of the performance reduction due to –fp-model precise –fp-model source was in the range of 12 to 15%. This was using the Intel Compiler 12.0 on an Intel Xeon® 5650 system with dual, 6-core processors at 2.67 GHz, 24GB memory, 12MB cache, running SLES* 10 x64 version with SP2.
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.

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 Intel compiler versions prior to 11.1, /Od (-O0) implied /Op (-mp), which could result in the generation of x87 instructions, even on Intel 64 architecture, unless overridden with /fp:source (-fp-model source).

From the 11.0 compiler onwards, loops containing math functions such as log() or sin() are not vectorized by default with /fp:precise (-fp-model precise), since this would result in a function call to a different math library that returns different, slightly less accurate results than the standard math library (see following section). The switch /Qfast-transcendentals (-fast-transcendentals) may be used to restore the 10.1 compiler behavior and re-enable vectorization.
Many routines in the Intel math libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Math Library Functions

As yet, no official 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 Mac 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 Mac 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 SSE2)
- By calls to the short vector math library (libsvml) for loops that can be vectorized

For the 11.0 and later compilers, 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
- Between one run-time processor and another. The math libraries contain function implementations that are optimized differently for different processors. 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 SSE3 instructions. The implementation that used SSE3 instructions would be invoked only on a processor that was known to support these.

  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 Intel-compatible

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

microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

The variations in the results of math functions discussed above are small. The expected accuracy, about 0.55 units in the last place (ulp) for the standard math library and less than 4 ulp for the short vector math library used for vectorized loops, is maintained both for different compiler releases and for implementations optimized for different processors.

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.

In the Intel compiler versions 11.1 and earlier, there is no switch that will override the processor dependency of results returned by math library functions. In version 12.0 of the compiler, (available in Intel® Parallel Composer 2011 and Intel® Composer XE), the switch /Qimf-arch-consistency:true (-fimf-arch-consistency=true) may be used to ensure bit-wise consistent results between different processor types of the same architecture, including between Intel processors and compatible, non-Intel processors. This switch does not ensure bit-wise consistency between different architectures, such as between IA-32 and Intel 64, or between Intel 64 and IA-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.

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.

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 Mac 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 Mac OS X).

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

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

Further Information

- Microsoft Visual C++* Floating-Point Optimization
  http://msdn2.microsoft.com/en-us/library/aa289157(vs.71).aspx
- The Intel® C++ and Fortran Compiler User and Reference Guides,
  "Floating Point Operations" section.
- "Floating Point Calculations and the ANSI C, C++ and Fortran Standard"
  http://www.intel.com/cd/software/products/asmo-na/eng/330130.htm
  or see the list at http://www.intel.com/cd/software/products/asmo-na/eng/330130.htm
- Goldberg, David: "What Every Computer Scientist Should Know About Floating-Point Arithmetic" *Computing Surveys*, March 1991, pg. 203

Appendix

Quick summary of primary floating-point switches:

Primary Switches	Description
/fp:*keyword* **-fp-model *keyword***	***fast**[=1|2], precise, except, strict, source* *[double, extended - C/C++ only]* *Controls floating point semantics*
/Qftz[-] -[no-]ftz	*Flushes denormal results to Zero*
Other switches	
/Qfast-transcendentals[-] -[no-]fast-transcendentals	*Enable[Disable] use of fast math 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)*
/Qfp-port -fp-port	*Round floating point results to user precision*
/Qprec -mp1	*More consistent comparisons & transcendentals*
/Op[-] -mp -[no]fltconsistency	*Deprecated; use /fp:precise etc instead*
/Qimf-precision:*name* -fimf-precision=*name*	*high, medium, low* *Controls accuracy of math library functions*
/Qimf-arch-consistency:*true* -fimf-arch-consistency=*true*	*Math library functions produce consistent results on different processor types of the same architecture*