Floating-point control in the Intel® C/C++ compiler and libraries
or
Why doesn't my application always give the same answer?

Martyn Corden
Developer Products Division
Software & Services Group
Intel Corporation
May 2013

Agenda

- Introduction
- Floating-point (FP) Model
  - Comparisons with gcc
- Performance impact
- Runtime math libraries
- Intel® Xeon Phi™ Coprocessors – what's different

Starting point

- The finite precision of floating-point operations leads to an inherent uncertainty in the results of a floating-point computation
  - Results may vary within this uncertainty
  - Usually, this is not a concern

- For some purposes, reproducibility beyond this uncertainty may be desired
  - Typically for reasons related to Quality Assurance

- “reproducible” is not necessarily more “accurate”

Floating Point (FP) Programming Objectives

- **Accuracy**
  - Produce results that are “close” to the correct value
    - Measured in relative error, possibly in ulp
- **Performance**
  - Produce the most efficient code possible
- **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

These options usually conflict!
Judicious use of compiler options lets you control the tradeoffs.
Different compilers have different defaults.

Why might you care about reproducibility?

- Parts of your analysis are run on different continents. Did everyone use the identical analysis program? Should they all see identical results?
- One institute used a different processor type and got a slightly different result. Was it the difference due to the different processor, or to a program difference?
- You rebuild a reconstruction program in debug mode to track down a problem. But the input data to the problem region have changed…
- Offline checks of an online trigger show tiny differences. Is there a hardware problem?

Agenda

- Introduction
- Floating Point (FP) Model
- Performance impact
- Runtime math libraries
- Intel® Xeon Phi™ Coprocessors – what's different

Floating Point Semantics

- The –fp-model switch lets you choose the floating point semantics at a coarse granularity. It lets you specify the compiler rules for:
  - **Value safety** (main focus)
  - FP expression evaluation
  - FPU environment access
  - Precise FP exceptions
  - FP contractions (fused multiply-add, "fma")
- Also pragmas in C99 standard
  - #pragma STDC FENV_ACCESS (ON|OFF) etc
- Old switches -mp (–float-consistency) now deprecated
  - Less consistent and incomplete; don't use

The –fp-model switch for icc

- **-fp-model**
  - fast [=1] allows value-unsafe optimizations (default)
  - fast=2 allows additional approximations
  - precise value-safe optimizations only
  - source | double | extended imply "precise" unless overridden
    see "FP Expression Evaluation" for more detail
  - except enable floating point exception semantics
  - strict precise + except + disable fma +
    don't assume default floating-point environment

- **-fp-model precise -fp-model source**
  - recommended for best reproducibility
  - also for ANSI/ IEEE standards compliance, C++ & Fortran
  - "source" is default with "precise" on Intel 64 Linux

GCC option

- -f[no-]fast-math is the high level option
  - It is off by default (different from icc)
  - It is turned on by -Ofast
- Components control similar features:
  - Value safety (-funsafe-math-optimizations)
    - includes reassociation
  - Reproducibility of exceptions
  - Assumptions about floating-point environment
  - Assumptions about exceptional values
- also sets abrupt/gradual underflow (FTZ)
- For more detail, check backup or http://gcc.gnu.org/wiki/FloatingPointMath

Value Safety

- In SAFE mode, the compiler may not make any transformations that could affect the result, e.g. all the following are prohibited.

x / x ⇔ 1.0	x could be 0.0, ∞, or NaN
x – y ⇔ - (y – x)	If x equals y, x – y is +0.0 while – (y – x) is -0.0
x – x ⇔ 0.0	x could be ∞ or NaN
x * 0.0 ⇔ 0.0	x could be -0.0, ∞, or NaN
x + 0.0 ⇔ x	x could be -0.0
(x + y) + z ⇔ x + (y + z)	General reassociation is not value safe
(x == x) ⇔ true	x could be NaN

- UNSAFE (fast) mode is the icc default
- VERY UNSAFE mode enables riskier transformations
  – (-fp-model fast=2)

Value Safety

Affected Optimizations include:

- Reassociation
- Flush-to-zero
- Expression Evaluation, various mathematical simplifications
- Approximate divide and sqrt
- Math library approximations

Reassociation

- Addition & multiplication are "associative" (& distributive)
  - a+b+c = (a+b) + c = a + (b+c)
  - a*b + a*c = a * (b+c)

- These transformations are equivalent ***mathematically***
  - but ***not*** in finite precision arithmetic

- Reassociation can be disabled in its entirety
  - ⇒ for standards conformance (C left-to-right)
  - Use **-fp-model precise**
  - May carry a significant performance penalty (other optimizations also disabled)

- Parentheses are respected only in value-safe mode!
  - -assume protect_parens compromise (Fortran only)

- See exercises for an example derived from a real app

Example (see exercises)

"tiny" is intended to keep a[i]>0

but… optimizer hoists constant expression (c+tiny) out of loop

tiny gets "rounded away" wrt c

icc –O1 reassoc.cpp; ./a.out

a = 0   b = inf

icc -fp-model precise reassoc.cpp;./a.out

a = 1e-20   b = 1e+20

g++ reassoc.cpp; ./a.out

a = 1e-20   b = 1e+20

g++ -O3 -ffast-math reassoc.cpp; ./a.out

a = 0   b = inf

```
#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";
}
```

Denormalized numbers and Flush-to-Zero (FTZ)

- Denormals extend the (lower) range of IEEE floating-point values, at the cost of:
  - Reduced precision
  - Reduced performance (can be 100 X for ops with denormals)

- If your application creates but does not depend on denormal values, setting these to zero may improve performance ("abrupt underflow", or "flush-to-zero",)
  - Done in SSE or AVX hardware, so fast
  - Happens by default at –O1 or higher (for icc, not gcc)
  - -no-ftz or –fp-model precise will prevent
    - Must compile main with this switch to have an effect
    - -fp-model precise –ftz to get "precise" without denormals
  - Not available for x87, denormals always generated
    - (unless trapped and set to zero in software – very slow)

- For gcc, -ffast-math or –Ofast sets abrupt underflow (FTZ)

Reductions

- Parallel implementations imply reassociation (partial sums)
  - Not value safe, but can give substantial performance advantage
  - -fp-model precise
    - disables vectorization of reductions, makes value safe
    - does not affect OpenMP* or MPI* or TBB reductions

      These remain value-unsafe (programmer's responsibility)

      - New features in Intel® Composer XE 2013

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

```
float Sum(const float A[], int n )
{
    float sum=0;
    for (int i=0; i<n; i++)
        sum = sum + A[i];
    return sum;
}
```

Run-to-Run Variations (for a single thread)

- same executable, input data, processor ➔ same result ?
  - **NO!** ("consistent within the expected uncertainty")
- Data alignment may vary from run to run, due to changes in the external environment
  - E.g. malloc of a string to contain date, time, user name or directory: size of allocation affects alignment of subsequent malloc's
  - Compiler may "peel" scalar iterations off the start of the loop until subsequent memory accesses are aligned, so that the main loop kernel can be vectorized efficiently
  - For reduction loops, this changes the composition of the partial sums, hence changes rounding and the final result
  - Occurs for both gcc and icc, when compiling for Intel® AVX
- To avoid, align data with _mm_malloc(size,32) (icc only)
  - or compile with –fp-model precise (icc) or without –ffast-math (larger performance impact)
- See exercise 14_run_to_run for an example

Reproducibility of Reductions in OpenMP*

- Each thread has its own partial sum
  - Partial sums are summed at end of loop
  - Breakdown, & hence results, depend on number of threads
  - Order of partial sums is undefined (OpenMP standard)
    - First come, first served
    - Result may vary from run to run (even for same # of threads)
    - For both gcc and icc
    - Can be more accurate than serial sum
  - For icc & ifort, option to define the order of partial sums (tree algorithm)
    - Makes results reproducible from run to run
    - export KMP_DETERMINISTIC_REDUCTION=yes (in 13.0)
      - May also help accuracy
      - Possible slight performance impact, depends on context
      - Requires static scheduling, fixed number of threads
      - Now default for large numbers of threads

FP Expression Evaluation

- In the following expression, what if a, b, c, and d are mixed data types (single and double for example)

a = (b + c) + d

Four possibilities for intermediate rounding, (corresponding to C99 FLT_EVAL_METHOD)

Indeterminate (default)		(-fp-model fast)
Use precision specified in source		(-fp-model source)
Use double precision	(C/C++ only)	(-fp-model double)
Use long double precision	(C/C++ only)	(-fp-model extended)

- Or platform-dependent default (-fp-model precise)
  - Defaults to –fp-model source on Intel64
  - Recommended for most purposes where “precise” is needed
- The expression evaluation method can significantly impact performance, accuracy, and portability

The Floating Point Unit (FPU) Environment

- FP Control Word Settings
  - Rounding mode (nearest, toward $+\infty$, toward $-\infty$, toward 0)
  - Exception masks, status flags
    (inexact, underflow, overflow, divide by zero, denormal, invalid)
  - Flush-to-zero (FTZ), Denormals-are-zero (DAZ)
  - x87 precision control (single, double, extended) [don't mess!]

- Affected Optimizations, e.g.
  - Constant folding (evaluation at compile time)
  - FP speculation
  - Partial redundancy elimination
  - Common subexpression elimination
  - Dead code elimination
  - Conditional transform, e.g.

    if (c) x = y; else x = z; ➔ x = (c) ? y : z;

FPU Environment Access

- When access disabled (default):
  - compiler assumes default FPU environment
    - Round-to-nearest
    - All exceptions masked
    - No FTZ/DAZ
  - Compiler assumes program will NOT read status flags

- If user might change the default FPU environment, inform compiler by setting FPU environment access mode!!
  - Access may only be enabled in value-safe modes, by:
    - **-fp-model strict** or
    - #pragma STDC FENV_ACCESS ON
  - Compiler treats control settings as unknown
  - Compiler preserves status flags
  - Some optimizations are disabled
- If you forget this, you might get **completely** wrong results!
  - Eg from math functions, if you change default rounding mode

Example

```
double  x.,  zero = 0.;
  feenableexcept (FE_DIVBYZERO);

  for( int i = 0; i < 20; i++ )
    x = zero ? (1./zero) : zero;
    …
```

Problem: F-P exception from (1./zero) despite explicit protection

- The invariant (1./zero) gets speculatively hoisted out of loop by optimizer, but the “?” alternative does not
- Compiler thinks safe, because exceptions are masked by default
- exception occurs before the protection can kick in
  - may not occur for Intel® AVX, which have masked vector instructions

Solution: Disable optimizations that lead to the premature exception

- icc –fp-model strict

  warns compiler that F-P defaults have been modified
- #pragma STDC FENV_ACCESS ON   does likewise
- icc –fp-speculation safe
  disables just speculation where this could cause an exception

Precise FP Exceptions

- When Disabled (default):
  - Code may be reordered by optimization
  - FP exceptions might not occur in the "right" places

- When enabled by
  -fp-model strict
  -fp-model except
  #pragma float_control(except, on)
  - The compiler must account for the possibility that any FP operation might throw an exception
    - Disables optimizations such as FP speculation
    - May only be enabled in value-safe modes
    - (more complicated for x87)
  - Does not unmask exceptions
    - Must do that separately, e.g.

    -fp-trap=common for C

    or functions calls such as feenableexcept()

    -fpe0 or ieee_set_halting_mode() for Fortran

Floating Point Contractions

- fused multiply-add (FMA) instructions may be generated on "Haswell" and on Intel® MIC architecture
  - Enabled by –fma or by default with –xcore-avx2 or -mmic
  - Disabled by –fp-model strict or –no-fma
    or #pragma fp_contract(off)
  - NOT disabled by –fp-model precise
  - -[no-]fma switch overrides –fp-model setting
  - icc does NOT support 4-operand AMD*-specific fma instruction
- When enabled:
- The compiler may generate FMA for combined multiply/add
  - Faster, more accurate calculations
  - Results may differ in last bit from separate multiply and add
- When disabled:
  - The compiler must generate separate multiply and add with intermediate rounding

Consequences of FMAs

- Results may change on “Haswell”
  - even without recompilation!
    - math library may take different path at run-time
- Recompilation can also convert intrinsics to FMA
- FMA breaks symmetry, e.g.:

```
double a,b,c,d,x;
c = -a;
d =  b;
x = a*b + c*d;
```

  - Without FMA, should evaluate to zero

    x = FMA(a, b, (c*d))   or   FMA (c, d, (a*b))

  - may not evaluate to zero
  - each has different rounding, unspecified which grouping the compiler will generate.

Agenda

- Introduction
- Floating Point (FP) Model
- Performance impact
- Runtime math libraries
- Intel® Xeon Phi™ Coprocessors – what's different

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

Typical Performance Impact of -fp-model source

- Measured on SPECCPU2006fp benchmark suite:
- -O2 or –O3
- Geomean reduction due to
  –fp-model precise –fp-model source
  in range 12% - 15% (on system with SSE instructions)
  may be more on systems with wider vectors

- With Intel® Compiler XE 2011 (12.0)
- Measured on Intel Xeon® 5650 system with dual, 6-core processors at 2.67Ghz, 24GB memory, 12MB cache, SLES* 10 x64 SP2

Use -fp-model source (/fp:source) to improve floating point reproducibility whilst limiting performance impact

Agenda

– Introduction
– Floating Point (FP) Model
– Performance impact
– Runtime math libraries
– Intel® Xeon Phi™ Coprocessors – what's different

Math Library Functions

- Different implementations may not have the same accuracy
  - On Intel 64:
    - libsvml for vectorized loops
    - libimf (libm) elsewhere
    - Processor-dependent code within libraries, selected at runtime
    - Inlining was important for Itanium, to get software pipelining, but less important for Intel 64 since can vectorize with libsvml
      - Used for some division and square root implementations
- No official standard (yet) dictates accuracy or how results should be rounded (except for division & sqrt)

-fp-model precise helps generate consistent math calls

- eg within loops, between kernel & prolog/epilog
- Remove or reduce dependency on alignment
- May prevent vectorization unless use –fast-transcendentals
  - When may differ from non-vectorized loop

More Math Library Features

- Select minimum precision
  - Currently for libsvml (vector); scalar libimf normally "high"

  -fimf-precision=<*high*|*medium*|*low*>
    - Default is off (compiler chooses)
    - Typically high for scalar code, medium for vector code
    - "low" typically halves the number of mantissa bits
      - Potential performance improvement
    - "high" ~0.55 ulp; "medium" < 4 ulp (typically < 2)
- -[no-]prec-div, -[no-]prec-sqrt
- -fimf-arch-consistency=<*true* | *false*>
  - Will produce consistent results on all microarchitectures or processors within the same architecture
  - Run-time performance may decrease
    - Since limits use of new instructions
  - Default is false (even with –fp-model precise !)

Math Libraries – potential issues

- Differences could potentially arise between:
  - Different compiler releases, due to algorithm improvements
    - Use –fimf-precision
    - another workaround, use later RTL with both compilers

  - Different platforms, due to different algorithms or different code paths at runtime
    - Libraries detect run-time processor internally
    - Independent of compiler switches
    - use -fimf-arch-consistency=true

  - Expected accuracy is maintained
    - 0.55 ulp for libimf
    - < 4 ulp for libsvml (default for vectorized loops)
- Adherence to an eventual standard for math functions would improve consistency but at a cost in performance.

Intel® Math Kernel Library

- Linear algebra, FFTs, sparse solvers, statistical, …
  - Highly optimized, vectorized
  - Threaded internally using OpenMP*
  - By default, repeated runs may not give identical results
- **C**onditional **B**it**W**ise **R**eproducibility (new)
  - Repeated runs give identical results under certain conditions:
    - Same number of threads
    - OMP_SCHEDULE=static (the default)
    - Same OS and architecture (e.g. Intel 64)
    - Same microarchitecture, or specify a minimum microarchitecture
    - Consistent data alignment*
  - Call `mkl_cbwr_set(MKL_CBWR_AUTO)` (run-to-run)
  - Call `mkl_cbwr_set(MKL_CBWR_COMPATIBLE)` (processors)
  - Or set environment variable `MKL_CBWR_BRANCH=…` (etc.)

Intel® Threading Building Blocks

- A C++ template library for parallelism
  - Dynamic scheduling of user-defined tasks
  - Supports parallel_reduce() pattern
  - Repeated runs may not give identical results

- For reproducible reductions:
  - **parallel_deterministic_reduce()** template function
  - In Intel® Composer XE 2013
  - Repeated runs give identical results provided the user-supplied body yields consistent results
    - Independent of the number of threads
      - Simple partitioner always breaks up work in the same way
    - But results may differ from a serial reduction
    - May be some impact on performance

Agenda

- – Introduction
- – Floating Point (FP) Model
- – Performance impact
- – Runtime math libraries
- – Intel® Xeon Phi™ Coprocessors – what's different

Floating-Point Behavior on the Intel® Xeon Phi™ Coprocessor

- Floating-point exception flags are set by Intel® Initial Multi-Core Instructions (Intel® IMCI)
  - the flags can be read, but unmasking & trapping are not supported
  - attempts to unmask will result in seg fault
  - -fpe0 (Fortran) and -fp-trap (C) are disabled
  - -fp-model except or strict will yield (slow!) x87 code that supports unmasking and trapping of floating-point exceptions
- Denormals are supported by Intel IMCI (but slow, like host)
  - Needs –no-ftz or –fp-model precise (like host)
- 512 bit vector transcendental math functions available (SVML)
  - Fast versions of division and square root with -fp-model fast=2
    - Sets –fimf-domain-exclusion to exclude special cases
  - Both SVML and fast inlined divide and sqrt sequences available
  - Fast versions of powers and logs to base 2
  - See Differences in floating-point arithmetic between Intel(R) Xeon processors and the Intel Xeon Phi(TM) coprocessor for details and status

Comparing Floating-Point Results between Intel® Xeon processors and the Intel® Xeon Phi™ Coprocessor

- Different architectures – expect some differences
  - Different optimizations
  - Use of fused multiply-add (FMA)
  - Different implementations of math functions
- To minimize differences (e.g. for debugging)
  - Build with –fp-model precise (both architectures)
  - Build with –no-fma (Intel® MIC architecture)
  - Select high accuracy math functions
    - (e.g. -fimf-precision=high; default with –fp-model precise)
  - Choose reproducible parallel reductions (slides 17,31,32)
    - Or run sequentially, if you have the patience…
  - Remember, the true uncertainty of your result is probably much greater!

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 Documentation, "Floating Point Operations"
- "Consistency of Floating-Point Results using the Intel® Compiler" http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/
- "Differences in Floating-Point Arithmetic between Intel® Xeon® Processors and the Intel® Xeon Phi™ Coprocessor" http://software.intel.com/sites/default/files/article/326703/floating-point-differences-sept11.pdf
- Goldberg, David: "What Every Computer Scientist Should Know About Floating-Point Arithmetic" Computing Surveys, March 1991, pg. 203

Legal Disclaimer

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

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, reference www.intel.com/software/products.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Cilk, Core Inside, FlashFile, i960, InstantIP, Intel, the Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

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

http://intel.com/software/products

Optimization Notice

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®

Software

–prec-div and –prec-sqrt Options

- Both override the –fp-model settings
- Default is –no-prec-sqrt, and somewhere between –prec-div and –no-prec-div

[-no]-prec-div /Qprec-div[-]

- Enables[disables] various divide optimizations
  - x / y ⇔ x * (1.0 / y)
  - Approximate divide and reciprocal

[-no]-prec-sqrt /Qprec-sqrt[-]

- Enables[disables] approximate sqrt and reciprocal sqrt

-[no-]fast-transcendentals

The compiler frequently optimizes calls of math library functions (like exp, sinf) in loops

- Uses SVML (short vector math library) to vectorize loops
- Uses the XMM direct call routines,
  e.g. exp → ___libm_sse2_exp   (IA-32 only)
  - May sometimes use fast in-lined implementations

This switch "-[no]fast-transcendental can be used to override default behavior

- Behavior related to settings of fp-model and other switches – see reference manual !!

gcc options

- -ffast-math implies
  - -fno-math-errno
  - -funsafe-math-optimizations
  - -ffinite-math-only
  - -fno-rounding-math
  - -fno-signaling-nans
  - -fcx-limited-range
  - & sets __FAST_MATH__
- -funsafe-math-optimizations implies
  - -fno-signed-zeros
  - -fassociative-math
  - -fno-trapping-math
  - -freciprocal-math
  - & sets abrupt underflow

Math Functions on the Intel® Xeon Phi™ Coprocessor

- Faster, more approximate versions of math functions can still be obtained with –fp-model precise by adding

  -fast-transcendentals –no-prec-div –no-prec-sqrt

  – See Differences in floating-point arithmetic between Intel(R) Xeon processors and the Intel Xeon Phi(TM) coprocessor for details and status

- Switches for finer control of math function accuracy:
  – -fimf-precision=<high|medium|low> [:func1,func2,…]
  – -fimf-max-error
  – -fimf-accuracy-bits
  – -fimf-absolute-error
  – -fimf-domain-exclusion

Math Functions on the Intel® Xeon Phi™ Coprocessor

- Math functions have special branches and code to handle "exceptional" arguments
  - Faster versions possible if this can be skipped
- -fimf-domain-exclusion= <value>; the bits of <value> indicate domains for which the compiler need not generate special code
  - 1 extreme values (close to singularities or infinities; denormals)
  - 2 NaNs
  - 4 infinities
  - 8 denormals
  - 16 zeros
  - E.g. -fimf-domain-exclusion=31 excludes all of these for all functions
- Can be restricted to specific functions, e.g.
  - -fimf-domain-exclusion=15:/sqrt,sqrtf gives fast, inlined versions of single & double precision square root
- -fp-model-fast=2 implies –fimf-domain-exclusion=15