Consistency of Floating Point Results
or
Why doesn't my application always give the same answer?

Martyn Corden

Developer Products Division
Software Solutions Group
Intel Corporation
March, 2008

Agenda

- Overview
- Floating Point (FP) Model
- Performance impact
- Runtime math libraries

Overview

- Some customers will not move to a new platform unless
  - Existing QA criteria are met
  - they can exactly reproduce the results from their old platform
  - Optimized builds exactly reproduce debug builds
- The right compiler options can deliver consistent, closely reproducible results whilst preserving good performance
  - Across IA-32, IA-64, Intel® 64 and compared to other IEEE-compliant platforms
  - Across optimization levels
  - Available in 9.1, 10.0 and 10.1 compilers

We Encourage use of -fp-model (/fp:) switches by customers for whom floating point consistency and reproducibility are important

Floating Point (FP) Programming Objectives

- **Accuracy**
  - Produce results that are "close" to the correct value
    - Measured in relative error, possibly in 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 platform to another
- **Performance**
  - Produce the most efficient code possible

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

Agenda

- Overview
- Floating Point (FP) Model
- Performance impact
- Runtime math libraries

Floating Point Semantics

- The –fp-model (/fp:) switch lets you choose the floating point semantics at a coarse granularity. It lets you specify the compiler rules for:
  - Value safety
  - FP expression evaluation
  - FPU environment access
  - Precise FP exceptions
  - FP contractions

Note – The remainder of the presentation uses the option spellings for Linux* and Mac OS* X. The concepts also apply on Microsoft* Windows*; see the compiler documentation for the corresponding spellings.

The –fp-model switch

- -fp-model
  - fast [=1] allows value-unsafe optimizations (default)
  - fast=2 allows additional approximations
  - precise value-safe optimizations only (also source, double, extended)
  - except enable floating point exception semantics
  - strict precise + except + disable fma

- Replaces –mp, -IPF-fltacc, etc
- -fp-model source is recommended for ANSI/ IEEE standards compliance (C++ & Fortran)

See http://www.intel.com/cd/software/products/asmo-na/eng/279090.htm
"Floating Point Calculations and the ANSI C, C++ and Fortran Standard"

Value Safety

- In SAFE (precise) mode, the compiler may not make any transformations that could affect the result,
  e.g. the following is prohibited:

$(x + y) + z \Leftrightarrow x + (y + z)$ general reassociation is not value safe

- UNSAFE (fast) mode is the default
  - The variations implied by "unsafe" are usually very tiny

- VERY UNSAFE (fast=2) mode enables riskier transformations

Examples of Value-Unsafe Optimizations

- Disabled by -fp-model precise:
  - reassociation, eg $(a+b) + c \rightarrow a + (b+c)$
  - zero folding eg $X+0 \rightarrow X$, $X*0 \rightarrow 0$
  - multiply by reciprocal eg $A/B \rightarrow A*(1/B)$
  - approximate sqrt
  - flush-to-zero
  - drop RHS to LHS precision, etc.

- FMA contractions are not disabled

Flush-to-zero (FTZ)

- -ftz and -no-ftz override the -fp-model settings
  - -ftz is implied by -O3 on IA-64, default on IA-32/Intel 64
- Sets [avoids setting] the hardware flush-to-zero mode
  - On IA-32, FTZ is only set after a successful runtime processor check
  - For IA-32/Intel 64, this only affects SSE code. There is no FTZ control for x87
    - available for both C and Fortran in 10.0
- Must compile main with this switch to have an effect
- FTZ is NOT a guarantee that denormals in a program are flushed to zero!! It is an optimization that ALLOWS denormals to be flushed to zero.

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 <u>not</u> in finite precision arithmetic

- Reassociation can be disabled in its entirety
  - $\Rightarrow$ standards conformance
  - Use **-fp-model precise**
  - May carry a significant performance penalty
    (other optimizations also disabled

  - -assume protect_parens   (Fortran only)
    - Respects the order of evaluation specified by parentheses

Reductions

- Parallel implementations imply reassociation (partial sums)
  - Not value safe
- -fp-model precise
  - disables vectorization of reductions
  - does not affect OpenMP* or MPI* reductions

    These remain value-unsafe (programmer's responsibility)

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

Example CAM (Community Atmosphere Model):

- Issue: Different results with/without optimization
  - Residuals increase by an order of magnitude

    - Cause: reassociation of

      A(I) + B + TOL -> A(I) + (B + TOL)

      when A(I)=-B >> TOL the result may become zero or very small – impact on final result can be large

    - Solution: -fp-model precise or

      -assume protect-parens with source change

      (A(I) + B) + TOL

Example WRF (Weather Research & Forecasting)

- Issue: different results when run on different numbers of processors under MPI
  - Cause: Loop bounds, and hence alignment, changes when number of MPI processes changes. Different code in loop kernel from that in prologue & epilogue can give different results for same data.
  - Solution: -fp-model precise to keep same code and library calls for loop prologue, epilogue and kernel.

Example WRF

- Issue: different results re-running the same binary on the same data on the same processor

  - Cause: with 10.x, global stack address and alignment can change due to external events. This changes number of iterations in prolog, & hence order of operations for vectorized reductions.

  - Solution: -fp-model precise to disable vectorization of reductions
    - In 11.x, the global stack is realigned consistently to a 128 byte boundary.

FP Expression Evaluation

- a = (b + c) + d

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

- 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 +∞, toward -∞, toward 0)
  - Exception masks (inexact, underflow, overflow, divide by zero, denormal, invalid)
  - Flush-to-zero (FTZ), Denormals-are-zero (DAZ)
  - x87 precision control (single, double, extended)
    - but beware of changing this!
- Status Flags
  - 1→1 mapping to exception masks

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

Precise FP Exceptions

- When Disabled (default):
  - Code may be reordered by optimization
  - FP exceptions might not occur in the "right" places
  - Especially important for x87 arithmetic

- When Enabled [by –fp-model strict, -fp-model except
  or #pragma float_control(except, on)]
  - The compiler must account for the possibility that any FP operation might throw an exception
    - Inserts fwait instructions for x87
    - Disables optimizations such as FP speculation
    - May only be enabled in value-safe modes
  - Does not unmask exceptions
    - Must do that separately (e.g. –fpe0 for Fortran)

Example

```
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;
```

Problem: FP exception from (1./zero) despite explicit protection

- The invariant (1./zero) gets speculatively hoisted out of loop by optimizer, but the "?" alternative does not
- exception occurs before the protection can kick in

Solution: Disable optimizations that lead to the premature exception

- icc –fp-model precise –fp-model except (or icc –fp-model strict)
  disables all optimizations that could affect FP exception semantics
- icc –fp-speculation safe
  disables just speculation where this could cause an exception
- #pragma floatcontrol around the affected code block (see doc)

Floating Point Contractions

- affects the generation of FMA instructions on IA-64
  - Enabled by default
  - Disabled by –fp-model strict or C/C++ #pragma
  - -[no-]IPF-fma switch overrides –fp-model setting

- When Enabled (default)
  - The compiler may generate FMA for combined multiply/add
  - Faster, more accurate calculations
  - Results may differ in last bit from separate multiply/add

- When Disabled [-fp-model strict, #pragma fp_contract(off)]
  - The compiler must generate separate multiply/add with intermediate rounding

Agenda

- Overview
- Floating Point (FP) Model
- Performance impact
- Runtime math libraries

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 SPECCPU2000 fp (base):
  - IA-64
    - ~20% for apps built with -O3 -ftz
  - Intel 64
    - ~1% for apps built with –O2 –ftz
    - ~15% for apps built with –fast –ftz
  - IA-32 using mostly SSE instructions
    - ~1% for apps built with –O2 –xW –ftz
  - IA-32 using mostly x87 instructions
    - Windows: ~5-10%
    - Linux: ~20-50% (due to many precision conversions)
- See "Floating Point Calculations and the ANSI C, C++ and Fortran Standard"

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

Known issues

- -fp-model source is needed even for debug builds (-O0)
  - In 10.x, -O0 implies -mp
  - Particularly important on Intel 64, where –O2 builds use SSE but –O0 builds use x87 because of implied –mp

- Even with –fp-model precise, the compiler may inline math functions or call optimized versions that may give different results
  - Use –nolib-inline (/Oi-) to prevent this
  - On IA-64, use –opt-report –opt-report-phase ipo_inl to show which functions get inlined.

- Vectorization results in calls to a different math library that yields different, slightly less accurate results than libm (libimf).
  - For exact comparison with debug builds,
  - can disable with –no-vec (10.x –vec-)

- -O0 –fp-model source –nolib-inline –no-vec

Agenda

- Overview
- Floating Point (FP) Model
- Performance impact
- Runtime math libraries

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 these libraries, dispatched at runtime
  - On IA-64:
    - Inlined code for many functions (to allow software pipelining)
    - libimf calls elsewhere

- No official standard (yet) dictates accuracy or how results should be rounded (except for division & sqrt)

- -fp-model precise helps generate consistent calls, eg within loops
    - Does not currently make vectorized loop consistent with non-vectorized

Math Libraries – known issues

- Differences could potentially arise between:
  - Different compiler releases, due to algorithm improvements
    - No workaround, except use later RTL with both compilers
  - Different platforms, due to different algorithms or different code paths at runtime
    - Libraries have internal processor dispatch
    - Independent of compiler switches
  - Expected accuracy is maintained
    - 0.55 ulp for libimf
    - < 4 ulp for libsvml (vectorized loops)
- Adherence to an eventual standard for math functions would improve consistency but at a cost in performance.

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"
- http://www.intel.com/cd/software/products/asmo-na/eng/279090.htm
  "Floating Point Calculations and the ANSI C, C++ and Fortran Standard"
- Goldberg, David: "What Every Computer Scientist Should Know About Floating-Point Arithmetic" *Computing Surveys*, March 1991, pg. 203

Summary/Call to Action

- Compiler options let you control tradeoffs between accuracy, reproducibility and performance

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

- Explain this to customers

Questions??

intel®

Leap ahead™

FPU Environment Access

- Affected Optimizations, e.g.
  - Constant folding
  - FP speculation
  - Partial redundancy elimination
  - Common subexpression elimination
  - Dead code elimination
  - Conditional transform, i.e.
    if (c) x = y; else x = z; ➔ x = (c) ? y : z;

Quick Overview of Primary Switches

Primary Switches	Description
/fp:keyword **-fp-model *keyword***	***fast****[=**1**|2], precise, source, except, strict* [double, extended - C++ only] Controls floating point semantics
/Qftz[-] -[no-]ftz	*Flushes denormal results to Zero*
Other switches	
/Qfp-speculation *keyword* -fp-speculation *keyword*	***fast****, safe, strict, off* *floating point speculation control*
/Qprec-div[-] -[no-]prec-div	*Improves precision of floating point divides*
/Qprec-sqrt[-] -[no-]prec-sqrt	*Improves precision of square root calculations*
/QIPF-fp-relaxed -IPF-fp-relaxed	*Same as -noprec-div –noprec-sqrt on IA-64*
/QIPF-fma[-] -[no-]IPF-fma	*Enable[Disable] use of fma instructions on IA-64*
/fpe:0 -fpe0	*Unmask floating point exceptions (Fortran only)*
/Qfp-port -fp-port	*Round floating point results to user precision*
/Qprec -mp1	*More consistent comparisons & transcendentals*
/Op[-] -mp [-nofltconsistency]	*Deprecated in 10.1; use /fp:source etc instead*