INTEL COMPILERS
A SUMMARY

Ronald W. Green

NOAA Training August 2019

Compiler Options for Analysis Tool

- Recommended:
  - -O2 –g –shared-intel -parallel-source-info=2
  - -debug inline-debug-info Enable the VTune Amplifier to identify inline functions and, according to the selected inline mode, associate the symbols for an inline function with the inline function itself or its caller. This is the default mode for GCC* 4.1 and higher, icc –O2 -g
  - -parallel-source-info=2. Enable/disable source location emission when OpenMP or auto-parallelism code is generated. 2 is the level of source location emission that tells the compiler to emit path, file, routine name, and line information.
- https://software.intel.com/en-us/vtune-amplifier-help-compiler-switches-for-performance-analysis-on-linux-targets

Software

FLOATING-POINT REPRODUCIBILITY WITH INTEL® PARALLEL STUDIO XE

Presenter :: Martyn Corden, Senior Software Engineer

Contributors :: Barbara Perz & Xiaoping Duan, Technical Consulting Engineers

Intel Corporation

Agenda

Introduction to Reproducibility

Reproducibility in Individual Tools (included in Intel® Parallel Studio XE)

- Intel® Compilers
  - Same executable
  - Different executables
- Intel® Performance Libraries
  - Intel® Math Kernel Library (Intel® MKL)
  - Threading Building Blocks (TBB)
- Intel® MPI Library

Summary

Introduction to Reproducibility

The finite precision of floating-point operations leads to **inherent uncertainty** in the results of a floating-point computation

- Results may vary within this uncertainty
  - At different optimization levels, on different processors …
  - Often, this is not concern

Some contexts require reproducibility beyond this uncertainty, including:

- Quality assurance, change control, functional safety, legal liability
  - E.g., financial services, automated driving, artificial intelligence, weather forecasting, crash simulation
- And this may come at a cost in performance

"Reproducible" is not necessarily more accurate!!!

Sources of Variability

Optimization is the primary source

- Can be processor-dependent (different instruction sets, SIMD widths, etc.)
- Parallel execution is a form of optimization
- Can rarely afford not to optimize a large application

Differences in accuracy

- Approximations, e.g. math functions, fast division & sqrt, flush denormals to zero, …
- Fused Multiply Add (FMA) instructions (more accurate than separate multiply add)

Changes in the order of operations

- Probably the most important source, especially for parallel applications
- Changes how rounding errors accumulate
- A different result doesn't necessarily mean less accurate!
  - Though users often consider the unoptimized result to be the "correct" one.

Changes in the Order of Operations

Examples:

- (x[i] + y) + z → x[i] + (y + z);
- a*b + a*c → a*(b+c)
- These transformations are exact mathematically
  - But not in finite precision arithmetic
- These examples affect sequential and parallel applications similarly
- For compiled code, these can be suppressed with compiler options

For a fuller discussion of the impact of compiler optimization on floating-point reproducibility of single threaded applications, including OpenMP* SIMD pragmas, see http://software.intel.com/articles/consistency-of-floating-point-results-using-the-intel-compiler/

Changes in the Order of Operations - Reductions

Summation as an example – also product, max, min, …

- Parallel implementations create partial results, then combine these
  - E.g. 1 result per SIMD lane or per OpenMP* thread or per MPI rank
- Result may differ from sequential evaluation
- But can give better performance

```
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 nt ) {
  float sum=0.;         // total sum
  float part_sum[nt]; // 1 partial sum per thread
  for (int it=0; it<nt; it++) part_sum[it]=0.;

  for (int it=0; it<nt; it++)  {
      for (int j=it*n/nt; j<(it+1)*n/nt; j++)
          part_sum[it] += A[j];
  }                     // increment partial sums
  for (int it=0; it<nt; it++) sum += part_sum[it];
                        // combine partial sums
return sum;  }
```

How to Make Parallel Reductions Reproducible (1)

Don't change the domain decomposition (composition of partial sums)

- SIMD: don't change the vector length; make independent of data alignment (-qno-opt-dynamic-align)
- OpenMP*: don't change number of threads
- Intel® TBB: use parallel_deterministic_reduce()
  - Decomposition into tasks is fixed, allocation of tasks to threads can vary
- Intel® MPI: don't change number of ranks

How to Make Parallel Reductions Reproducible (2)

Don’t change the order in which partial sums are combined

- SIMD: automatic
- OpenMP* threading: standard allows any order
  - Default first come, first served for low thread counts
  - KMP_DETERMINISTIC_REDUCTION=yes with static scheduling for reproducibility
  - Default for larger thread counts
- Intel® TBB: use parallel_deterministic_reduce()
  - Partial sums are combined in a predefined order
- Intel® MPI: choose a topology-unaware reduction algorithm (see later)

REPRODUCIBILITY IN THE INTEL® COMPILER

Same executable; Different executables

Same Executable: Run-to-Run Variations (single thread)

If I rerun the same executable on the same processor with the same input, surely I must get the same result ?

- **NO!!** (“consistent within expected FP uncertainty”)

Data alignment may vary from run to run, due to changes in external environment

- May change which loop iterations are executed in the vectorized kernel
- Iterations in peel & remainder loops may be optimized differently than the main kernel
- For reduction loops, the partial sums change and hence the rounding and final result

To avoid dependency on alignment, compile with -qno-opt-dynamic-align

- Less performance impact than -fp-model precise or consistent

Or align data, e.g. _mm_malloc(size,64) (C/C++), -align array64byte (Fortran), etc.

Run-to-Run Variations (example on Linux*)

```
        gettimeofday(&start, NULL);
        na = (int)start.tv_sec%16;    //  size of a depends on time of day
        a = (float *) malloc(4*na);     //  variable size of a changes alignment of b
        b = (float *) malloc(4*nb)
        …
//    ***********  vectorized  reduction  ***********
        for (int i=0; i<nb; i++) sum = sum + b[i];    // order of additions depends on alignment
```

```
> icc  -xavx  -o run_to_run_icc  run_to_run.cpp;  ./run_to_run.sh   // run once per second
time in secs %16 =  8   start address = 0x609950   alignment = 16   sum = -8.819996
…
time in secs %16 = 12  start address = 0x609960   alignment = 32   sum = -8.82
…
time in secs %16 =  0   start address = 0x609930   alignment = 48   sum = -8.819996
…
time in secs %16 =  4   start address = 0x609940   alignment = 0    sum = -8.82
```

In practical examples, a string was malloc-ed to hold the date, time, username or run directory name, not necessarily in user code

Same Executable: Different Processor

Main difference comes from run-time dispatch in the math library

- Different function implementations on different processors
  - E.g. using FMAs on Intel® AVX2 and higher
  - May be more accurate (and different!) as well as better performing
- For consistent math function results, use -fimf-arch-consistency=true
  - Along with -qno-opt-dynamic-align
  - Some cost in performance, since can't use newer features such as FMA

Improved reproducibility in 18.0 and later compilers

- No run-time dispatch in libsvml if targeting Intel® AVX (-xavx) or higher
  - Use -fimf-use-svml instead of -fimf-arch-consistency=true
  - May perform better

Same Executable: Different Domain Decomposition

Different numbers of threads and/or processes

- Loop parameters, e.g. iteration counts, may vary
  - Changes data alignment
  - Changes what is computed in vector kernel and what in peel/remainder loops

For good consistency and good performance:

- No standard OpenMP* or MPI reductions
- Suppress dependency on alignment with -qopt-no-dynamic-align
- -fimf-use-svml for consistent math functions between remainder & kernel
- Remainder loop and vector kernel are optimized independently, so might still sometimes require -fp-model precise or -fp-model consistent

Different Executables

For reproducibility between different optimization levels on same processor:

-fp-model precise -no-fma   (/fp:precise /Qfma-)

For best floating-point reproducibility between different optimization levels and different processor types, use:

-fp-model consistent (/fp:consistent)

- Equivalent to -fp-model precise -fimf-arch-consistency=true -no-fma

With the version 18 or 19 compiler, may try adding -fimf-use-svml (/Qimf-use-svml)

- Re-enables vector math functions while preserving reproducibility
- May reduce impact on performance
- Results may be very slightly less accurate

Reproducibility Between Different Operating Systems?

What about consistency between Windows* & Linux*?

- Not something explicitly tested or supported currently
  - No intentional differences in Intel’s math run-time libraries
  - Known differences in vectorization lead to different math library calls
    - libm and SVML
  - Can work around this with /Qimf-use-svml (-fimf-use-svml) in 18.0 and later

- Best shot: /fp:consistent /Qimf-use-svml and -fp-model consistent -fimf-use-svml
  - May aid porting from Windows to Linux
  - Work in progress …

Reproducibility Between Different Intel® Compiler Versions?

Not necessarily, even with -fp-model precise (/fp:precise)

- Results from compiler-generated code should not change
- Results of math functions may change
  - Usually, to improve accuracy, rather than performance
  - Expected accuracy is maintained:
    - 0.6 ulp for libm (libimf)
    - < 4 ulp for libsvml (default for vectorized loops) or < 1 ulp for high precision
- Try -fp-model consistent (/fp:consistent)
  - Excludes new instructions such as FMAs
  - Else -fimf-precision=high (/Qimf-precision:high) to minimize differences
  - Another workaround: use the later run-time library with both compilers

ulp = Unit in the Last Place, effectively 1 bit

Adherence to an eventual standard for math functions would improve consistency but might affect performance.

REPRODUCIBILITY IN THE INTEL® PERFORMANCE LIBRARIES

Intel® MKL and TBB

Intel® Math Kernel Library (Intel® MKL)

Linear algebra, FFTs, sparse solvers, statistical, …

- Highly optimized, vectorized
- Threaded internally using OpenMP* or TBB
- By default, repeated runs on same processor may not give identical results
  - Due to variations in the order of operations
- Repeated runs on different processors may result in additional variations
  - MKL functions are "dispatched"
    - They detect the processor/microarchitecture on which they are running
    - execute a code path optimized for that microarchitecture

Intel® MKL Conditional Numerical Reproducibility

Repeated runs give identical results under certain conditions:

- Use OpenMP* version, not TBB
- Same number of threads
- OMP_SCHEDULE=static (the default)
- OMP_DYNAMIC=false and MKL_DYNAMIC=false (or unset)
- Same OS and architecture (e.g. Intel 64 Linux*)
- Same microarchitecture, or specify a minimum microarchitecture
  - Call `mkl_cbwr_set(MKL_CBWR_AUTO)` (run-to-run, same processor)
  - Call `mkl_cbwr_set(MKL_CBWR_COMPATIBLE)` (all processors)
  - Call `mkl_cbwr_set(MKL_CBWR_AVX2)` (will give identical results on processors code named Haswell, Knights Landing & Skylake, but not Sandy Bridge or Nehalem)
  - Or set environment variable `MKL_CBWR_BRANCH=MKL_CBWR_…`
    - `COMPATIBLE, SSE2, AVX, AVX2, AVX512, AVX512_MIC, …`

Reproducibility Between Different Processors

Result depends on ISA target (same color ⇒ same result)

- If ISA unsupported, defaults to AUTO

ISA target: Processor Codename	AUTO	AVX512 -MIC	AVX512	AVX2	AVX	SSE2 or compatible
Knights Landing	🟧	🟧	🟧	🟦	🟨	🟥
Skylake	🟩	🟩	🟩	🟦	🟨	🟥
Haswell	🟦	🟦	🟦	🟦	🟨	🟥
Sandy Bridge	🟨	🟨	🟨	🟨	🟨	🟥
Compatible non-Intel processor	🟥	🟥	🟥	🟥	🟥	🟥

Estimated Impact of Target ISA on Performance

Reproducibility between processors may come at a cost!

Targeted ISA	Estimated Relative Performance
MKL_CBWR_AUTO	1.0
MKL_CBWR_AVX512	1.0
MKL_CBWR_AVX2	0.50
MKL_CBWR_AVX	0.27
MKL_CBWR_SSE2	0.12

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Configuration: Measured using DGEMM from Intel® Math Kernel Library 2018 with 3 threads for 4999x4999 matrices on an Intel® Xeon® Platinum 8180M system with dual, 28-core 2.50GHz processors with 32 GB RAM, 38MB cache, OS Fedora 25.

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
- Repeated runs of the same binary give identical results provided the user-supplied body yields consistent results (see compiler section)
  - Independent of the number of threads
    - Simple partitioner always divides up work in the same way
  - But results may differ from a serial reduction
  - May be some impact on performance

REPRODUCIBILITY IN THE INTEL® MPI LIBRARY

Intel® MPI Conditional Numerical Reproducibility

Reproducibility wrt distribution of ranks over nodes, provided that:

- Number of ranks (processes) remains constant
- Microarchitecture (processor type) does not change
- Compiled code within each rank is reproducible

Collective operations (e.g. global reductions such as MPI_Reduce())

- Must use reproducible implementations
- Algorithm must not be “topology aware”
  - i.e. must not be sensitive to the distribution of ranks amongst nodes

Algorithms may be specified by environment variables:

I_MPI_ADJUST_REDUCE, _ALLREDUCE, _REDUCE_SCATTER, _SCAN, _EXSCAN, …

Algorithms for MPI_REDUCE() in Intel® MPI

	Rank distribution algorithm	0246 node1 1357 node2	All node1	0145 node1 2367 node2	0123 node1 4567 node2
1	Shumilin's	🟦	🟦	🟦	🟦
2	Binomial	🟥	🟥	🟥	🟥
3	Topology aware Shumilin's	🟨	🟦	🟨	🟧
4	Topology aware binomial	🟨	🟥	🟨	🟥
5	Rabenseifner's	🟥	🟥	🟥	🟥
6	Top. aware Rabenseifner's	🟨	🟥	🟨	🟥
7	K-nomial	🟩	🟩	🟩	🟩

same color means same result

Two nodes of Intel® Core™ i5-4670T at 2.30 GHz, 4 cores & 8 GB memory each, one running Red Hat EL 6.5, the other running Ubuntu 16.04. The sample code from "Intel® MPI Library Conditional Reproducibility" in Parallel Universe Magazine Issue 21 (see references) was used.

“Reproducibility” Means Different Things to Different People!

The conditions under which floating-point results can be reproduced exactly are different for different tools:

- Intel® Compiler
  - Same or different binary, processor type, optimization level
  - Same number of threads, static scheduling (for OpenMP*)
- Intel® MKL
  - Same binary, same # of threads, static scheduling, different processor types
- TBB
  - Same binary, different numbers of threads, different processor types
- Intel® MPI
  - Same binary, same # of ranks, same processor types and run-time environment, different distribution of ranks amongst nodes

Further Information

"Consistency of Floating-Point Results using the Intel® Compiler"
  http://software.intel.com/articles/consistency-of-floating-point-results-using-the-intel-compiler/

The Intel® MKL Developer Guide, section "Obtaining Numerically Reproducible Results"
  https://software.intel.com/mkl-linux-developer-guide

"Intel® MPI Library Conditional Reproducibility", Parallel Universe Magazine Issue 21,
  https://software.intel.com/sites/default/files/managed/75/20/parallel_mag_issue21.pdf

Tuning the Intel MPI Library: Basic Techniques", section "Tuning for Numerical Stability"
  https://software.intel.com/articles/tuning-the-intel-mpi-library-basic-techniques

The Intel® C++ and Fortran Compiler Developer Guides, section "Floating-Point Operations"
  https://software.intel.com/fortran-compiler-developer-guide-and-reference
  https://software.intel.com/cpp-compiler-developer-guide-and-reference

Legal Disclaimer & Optimization Notice

What If…?

For certain input data, application gets different results on Sandy Bridge and Haswell processors.

- Mixed C and Fortran
- Uses MKL with FFTW.
- Built with -O0 -fp-model consistent -qno-opt-dynamic-align -no-fma
- For MKL: export MKL_CBWR=COMPATIBLE

What could be wrong?

What Might Be Wrong?

Misspelling MKL_CBWR_BRANCH ?

If the misspelling on previous slide was just a typo, then:

- Some code (e.g. in a library) was linked that had not been compiled with -fp-model consistent;
- Using the version of MKL layered on TBB instead of on OpenMP*;
- Something overriding static OpenMP scheduling;
- Not specifying the number of threads explicitly, with OMP_NUM_THREADS (or MKL_NUM_THREADS or MKL_DOMAIN_NUM_THREADS).
  OpenMP defaults to the maximum number of available hardware threads;
  Sandy Bridge and Haswell processors probably have different numbers of cores.

intel®

experience what's inside™