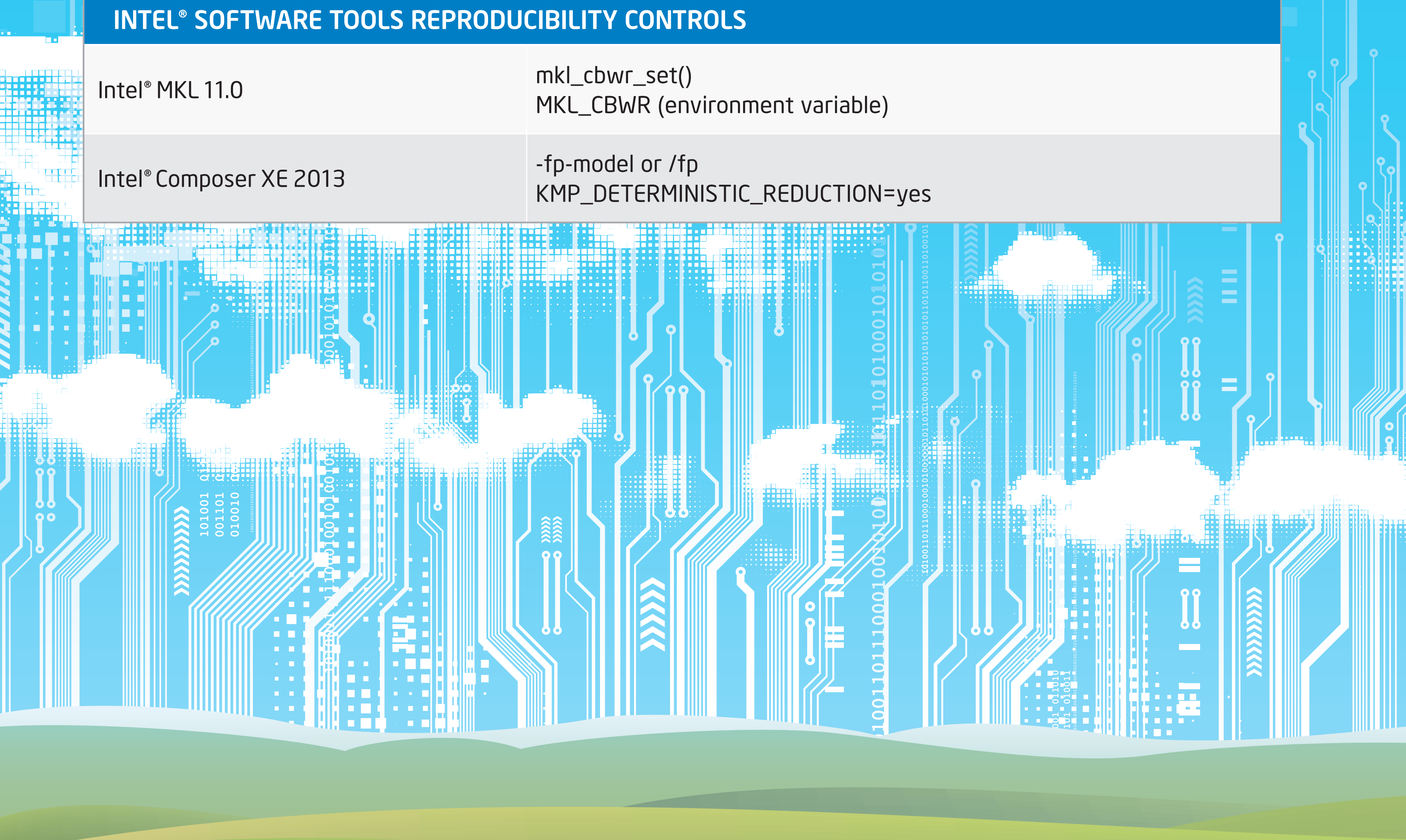
Using the Intel® Math Kernel Library (Intel® MKL) and Intel® Compilers to Obtain Run-to-Run Numerical Reproducible Results

by Todd Rosenquist, *Technical Consulting Engineer, Intel® Math Kernal Library*
and Shane Story, *Manager of Intel® MKL Technical Strategy, Intel Corporation*

Floating-point applications from Hollywood to Wall Street have long faced the challenge of providing both great performance and exactly the same results from run to run, or in other words, reproducible results. While the main factor causing a lack of reproducible results is the non-associativity of most floating point operations, there are other contributing factors such as runtime, selectable optimized code paths, non-deterministic threading and parallelism, array alignment, and even the underlying hardware floating-point control settings.

In this article for Intel® software tool users and programmers, we outline how to use the Intel® Math Kernel Library (Intel® MKL) and Intel® compiler features to balance performance with the reproducible results applications require. These new reproducibility controls in Intel® Parallel Studio XE 2013 help make consistent results from run to run possible:

INTEL® SOFTWARE TOOLS REPRODUCIBILITY CONTROLS	
Intel® MKL 11.0	mkl_cbwr_set() MKL_CBWR (environment variable)
Intel® Composer XE 2013	-fp-model or /fp KMP_DETERMINISTIC_REDUCTION=yes



After many years of seeing software performance increase with processor clock speed, the last half-decade has seen the flattening of clock rates and the increasing availability of multicore systems. With each successive generation of microprocessors, improvement in software performance requires the use of newly added instructions to exploit the capabilities of the processor, as well as threaded algorithms designed to leverage the growing number of computational cores. To keep up with these changes, many developers turn to software tools. Optimizing compilers exploit opportunities for instruction and data-level parallelism and can automatically thread computationally intensive portions of a program. Software libraries provide tools to thread your code or allow you to extract parallelism automatically through calls to highly optimized, threaded functions. Many software programmers have adopted and use these high performance tools to extract greater levels of performance. In doing so, the likelihood of generating inconsistent results from run to run has grown.

Let's consider two scenarios. Artists in animation studios work every day with advanced modeling tools that allow them to move their actors through a virtual world. These modeling tools include physics engines that can simulate the real-world behavior of clothes, hair, or fluids, and therefore will naturally use floating-point models similar to those used in science and engineering applications. While accuracy and precision may not always be the first concern, especially in early stages of the process, getting the same results can be of the utmost importance. If a cloak follows a slightly different trajectory each time the artist runs through a multi-second sequence, the artist has lost some control over the creative process. Which trajectory will be used when the scene goes through further rendering and post-processing steps? The problem would be compounded by the fact that a single scene may have many such models that may interact to produce completely unpredictable results.

A second scenario involves mathematicians on Wall Street who develop algorithms for various applications from options pricing to risk analysis. In this field, getting results quickly means money—and sometimes a lot of money. The "quants" who develop these algorithms are faced with a balancing act between getting the answer quickly and the simulation time required to provide the most reliable answer. An increase in the performance of an algorithm can mean a decision sooner or a better decision in the same amount of time—a win in either case. However, optimized floating-point calculations that are a part of these models can often introduce rounding error. This means that if an earlier decision must be revisited and the model run again, it is possible that the result might be slightly different. The uncertainty can result in questions or issues later that programmers would prefer to avoid.

These are just two of many scenarios[1] encountered over the last few years by users of Intel MKL. This is a popular library of highly optimized parallel floating-point math functions that has been successfully used by customers in many application areas for over 15 years. For application programmers who demand reproducible results, there have not been any guarantees and only the limited option of running a sequential version of the library.

"Floating-point applications from Hollywood to Wall Street have long faced the challenge of providing both great performance and exactly the same results from run to run, or in other words, reproducible results."

So, what exactly is the reproducibility problem? The issue is rooted in the way floating-point numbers are represented, the order in which they are operated on by the computer, and the rounding errors that may be introduced. It is a well-known fact that for general floating-point numbers represented in an IEEE single or double precision format[2], the mathematical associative property does not in general hold.[3] In simpler terms, $(a + b) + c$ may not equal $a + (b + c)$.

It may help to consider a specific example. With pencil and paper, $2^{-63} + 1 + -1 = 2^{-63}$. If, instead we do this same computation on a computer using double precision floating-point numbers, we get $(2^{-63} + 1) + (-1) \approx 1 + (-1) = 0$ since $(2^{-63} + 1)$ rounds to 1, or possibly $2^{-63} + (1 + (-1)) \approx 2^{-63} + 0 = 2^{-63}$ through a slight modification in the order of operations. Clearly 0 does not equal 2^{-63}, so the order of operations not only influences how and when rounding occurs but also the final computed result. Compilers typically refer to this ordering ambiguity as re-association.

Introducing application-level parallelism further increases the likelihood of producing nonreproducible results. The reason is a direct carryover from the order of operations argument just described. Whenever work is distributed among multiple threads or processes, any change in the order of operations within a computational dependency chain may result in a difference not only in the intermediate results, but also in the final computed results. Straightforward array element sum and product reduction operations are simple examples when the array elements have been distributed across multiple threads; partial sums or products are computed and then combined across threads into a single value. Any change in how the arrays are distributed, or the order in which a thread-specific sum or product is combined with another, may influence the final reduced sum or product. More broadly, how to handle parallelism in a consistent and predictable way falls under the category of deterministic parallelism.[4]

When you consider that a typical application may do millions of floating-point operations, it becomes readily apparent how the order of operations influences the final computed results.

Intel Math Kernel Library

Intel MKL 11.0 introduces Conditional Numerical Reproducibility functions to help users obtain reproducible floating-point results from Intel MKL functions under certain conditions.[5] When using these new features, Intel MKL functions are designed to return the same floating-point results from run to run, subject to the following limitations:

- Input and output arrays in function calls must be aligned on 16-, 32-, or 64-byte boundaries on systems with SSE/AVX1/AVX2 instructions support respectively.
- Control over the number of threads must remain the same from run to run for the results to be consistent.

The application-related factors within a single executable program that affect the order in which floating-point operations are computed include code path selection based on runtime processor dispatching, data array alignment, variation in number of threads, threaded algorithms, and internal floating-point control settings. Up until now, users were unable to control the library's runtime dispatching and how its functions were internally threaded. However, they were able to manage the number of threads, check the floating-point settings, and take steps to align memory when it is allocated.[6]

Intel MKL does runtime processor dispatching in order to identify the appropriate internal code paths to traverse for the Intel MKL functions called by the application. The code paths chosen may differ across a wide range of Intel® processors and IA-compatible processors, and may provide varying levels of performance. For example, an Intel MKL function running on an Intel® Pentium® 4 processor may run an SSE2-based code path. On a more recent Intel® Xeon® processor supporting Intel® Advanced Vector Extensions (AVX) that same library function may dispatch to a different code path that uses AVX instructions. This is because each unique code path has been optimized to match the features available on the underlying processor. This feature-based approach to optimization, by its very nature, amplifies the reproducibility challenges already described. If any of the internal floating-point operations are done in a different order, or are re-associated, then the computed results may differ.

"Increasingly, with each successive generation of microprocessors, improvement in software performance requires the use of newly added instructions to exploit the capabilities of the processor, as well as threaded algorithms designed to leverage the growing number of computational cores."

Using the Reproducibility Features

Intel MKL 11.0 includes new functions and environment variables (shown in the table) designed to help users get numerical reproducible results from the Intel MKL functions used. These functions and variables allow users to control what code paths are executed and ensure deterministic thread execution. The greater the number of unique processors a given Intel MKL-based application needs to support with reproducible results, the tighter the restriction on which instructions can be used by the common code path executed within the library. For reproducible results across all processors supported by Intel MKL, specify that the "COMPATIBLE" code path be used. Because this code path uses instructions common across all IA-compatible processors, users should not expect to see the same levels of performance that can be achieved on the latest processors. If on the other hand, your processors are limited to more recent generations of Intel® processors, then a code path can be chosen that uses the latest instruction sets and therefore provides greater performance. The table outlines the trade-off between breadth of compatibility and performance for a number of the code paths found in Intel MKL.

Notes:

- Ensure your application uses a fixed number of threads and aligns input and output arrays for Intel MKL function calls.
- On non-Intel CPUs, the results may differ because the MKL_CBWR_COMPATIBLE code-path is run instead.
- The implementation of approximation instructions (e.g., rcpss/ps, rsqrtss/ps) in Intel processors may differ with the implementations from other vendors and may return different results. The COMPATIBLE setting ensures that Intel MKL uses an SSE2-only code-path which does not use these instructions.

Performance Implications

Dispatching optimized code paths based on the capabilities of the processor on which it is running is central to the optimization approach used by Intel MKL, so it is natural that there should be some performance trade-offs when requiring consistent results. If limited to a particular code path, Intel MKL performance can in some circumstances degrade by more than half. This can be easily understood by noting that matrix-multiply performance nearly doubled with the introduction of new processors supporting AVX instructions, primarily because the internal vector register width also doubled. In other cases, performance may degrade by between 10 percent and 20 percent if the algorithms are merely constrained to maintain the order of operations.

Maximum Compatiblity

For consistent results ...	Function Call mkl_cbwr_set(...)	Environment Variable MKL_CBWR=
on Intel® or Intel®-compatible CPUs supporting SSE2 instructions or later	MKL_CBWR_COMPATIBLE	COMPATIBLE
on Intel® processors supporting SSE2 instructions or later	MKL_CBWR_SSE2	SSE2
on Intel processors supporting SSE4.2 instructions or later	MKL_CBWR_SSE4_2	SSE4_2
on Intel processors supporting Intel® AVX or later	MKL_CBWR_AVX	AVX
from run to run (but not processor-to-processor)	MKL_CBWR_AUTO	AUTO

Maximum Performance

Intel® Composer XE 2013

This new feature in Intel MKL ensures that in many circumstances you can get excellent performance, while still meeting your reproducibility requirements. It is worth noting that a comprehensive solution that ensures reproducible results across your entire application will require that the other tools used to build your application also provide reproducible results. The optimizing compilers in Intel Composer XE 2013 are part of Intel's broader reproducibility solution and provide compilation options to ensure that compiler-generated code produces reproducible results from run to run.

For example, the -fp-model compilation switches on Linux* (or /fp on Windows*) provide options for controlling value safety, floating-point expression evaluation, floating-point unit environment access, precise floating-point exceptions, and handling floating-point contractions. As with Intel MKL, these controls will affect the ability of the compiler to optimize code using re-association, expression evaluation, divide and sqrt, and other math library approximations.[7] The Intel compiler also provides threading via the OpenMP* model. With the latest compiler, the reduction stage in threaded parallel sections can be forced to provide reproducible results by setting KMP_DETERMINISTIC_REDUCTION=yes.

Let's consider the case where you would like the resulting executable to produce the same results on the following three systems:

> One 4-core processor supporting SSE 4.2 instructions
> One 4-core processor supporting AVX instructions
> Two 4-core processors supporting AVX instructions

First, to ensure that Intel MKL functions provide reproducible results, we must make sure all arrays are aligned on 16-byte boundaries (the mkl_alloc() function suits this purpose). Then, ensure that the number of threads used on each system remains fixed and does not vary during the run. By default, Intel MKL uses as many threads as there are cores, so you should fix the number of threads at four using the mkl_set_num_threads() function. Finally, because, Intel MKL dispatches an optimized code path based on the instruction set available, you need to use the new reproducibility control to configure this: mkl_cbwr_set(MKL_CBWR_SSE4_2. You will then also need to use an appropriate –fp-model (or /fp) flag to ensure the compiler returns consistent results. Applications threaded using OpenMP should also specify a fixed number of threads (omp_set_num_threads(4)) and set KMP_DETERMINISTIC_REDUCTION=yes.

Conclusion

The features described here are intended to help programmers generate reproducible results within their applications under a manageable set of constraints, such as a fixed number of threads, when running on the same operating system, if the underlying processor family stays the same, only when input/output arrays are aligned, and so on. Looking forward, the plan is to continue to explore ways to lessen the constraints and extend product features to incorporate the latest algorithms that ensure deterministic parallelism on a variable number of threads, to remove the input/output alignment restrictions, and to use automated analysis tools to validate that the code generated produces the same results regardless of the underlying operating system or processor.

Bibliography

1. The Limits of Reproducibility in Numerical Simulation, Kai Diethelm, Computing in Science & Engineering, 2012.
2. IEEE Standard for Binary Floating Point Arithmetic, ANSI/IEEE Standard No. 754, American National Standards Institute, Washington, DC, 1988.
3. What Every Computer Scientist Should Know about Floating-Point Arithmetic, David Goldberg, ACM Computing Surveys 23, 5-48, 1991.
4. Is Parallel Programming Hard, Stephen Lewin-Berlin, Intel® Software Network, http://software.intel.com/en-us/articles/is-parallel-programming-hard-1/, 2009.
5. The Flagship High-Performance Computing Math Library for Windows*, Linux*, and Mac OS* X. Intel® Math Kernel Library (Intel® MKL) 10.3, Intel Software Network, http://software.intel.com/en-us/articles/intel-mkl/.
6. Getting Reproducible Results with Intel® MKL, Todd Rosenquist, Intel® Software Network, http://software.intel.com/en-us/articles/getting-reproducible-results-with-intel-mkl/, 2010.
7. Consistency of Floating-Point Results using the Intel® Compiler, Martyn Corden, Intel® Software Network, http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/, 2010.

"The application-related factors that affect the order in which floating-point operations are computed within a single executable program include code path selection based on runtime processor dispatching, data array alignment, variation in number of threads, threaded algorithms, and internal floating-point control settings."

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