Cray X1 Tuning for Performance Using Compilation Options

Terry Greyzck, *Cray Inc.*

ABSTRACT: *The Cray X1 compilers' command-line options and source code directives provide significant user control over program optimization. This paper describes how common options and directives can influence specific optimizations, affect compile time, and sometimes cause unexpected side effects.*

KEYWORDS: Cray, X1, multistreaming, shared-memory parallelism, vectorization

1. Introduction

1.1. The X1 Compilers

The high level languages supported for the Cray X1 series of computers are Fortran, C and C++. The compilers for these languages provide a high level of optimization, while adhering to the appropriate standards.

All Cray X1 compilers share a common optimizer and code generator. Sharing this technology allows for consistent optimization capabilities and reliability across different source code languages.

1.2. Default Optimization

The default command line optimization (that is, no specified options) extracts vector and multistream levels of parallelism from the code, in addition to the lower levels of parallelism found in all compilations. For Fortran, this default optimization is roughly the same as specifying:

-Oscalar2 –Ovector2 –Ostream2 –Oipa3 –Ofp2

with similar options for C and C++. The default optimization provides, at a minimum, the following:

- Somewhat aggressive scalar optimization
- Automatic SIMD parallelism (vectorization)
- Automatic MIMD parallelism (multistreaming)
- Automatic limited interprocedural optimization
- Aggressive floating point optimizations

This is clearly a very high level of optimization. There are also a large number of compiler directives available to control fine-grain optimization. These directives are source code annotations that have meaning to the compiler, and are documented in the appropriate manuals and optimization guides.

1.3. Tuning Through the Command Line

The default optimization is very aggressive, yet it strikes a balance between code safety, consistent results between runs (and consistency as compared to other vendors), and optimized compile times. If you want maximum performance when running on the Cray X1, and you can accept additional compilation time, and slightly different (albeit correct) floating-point results, then you should consider introducing command line options to enhance optimization.

For any given application, the tuning of individual options may have anywhere from no performance impact, to a dramatic performance impact. The performance impact on your application may vary.

(In the examples below, the Fortran option is presented first, followed by the C and C++ equivalent options. The options for the C and C++ compilers are identical unless otherwise specified.)

2. Increasing Automatic Optimization Levels

2.1. -O3

This is the highest level of optimization that can be specified using the *-O* option, and is roughly equivalent to:

Fortran:

-Oscalar3 –Ovector3 –Ostream3

C and C++:

-hscalar3 –hvector3 –hstream3

-O3 is a convenient shortcut to specifying all of these options individually, as described in the following sections.

2.2. -Oscalar3; -hscalar3

For the Cray X1 compiler, *scalar optimization* refers to all optimizations that do not directly affect parallelization. However, these optimizations can have a large impact on parallel code, primarily due to Amdahl's law. These optimizations include, but are not limited to:

- Address computation optimizations
- Array syntax simplification
- Branch reduction
- Common subexpression elimination
- Data extraction
- Dead code elimination
- Idiom recognition
- Index range splitting
- Last value capture
- Loop invariant hoisting
- Loop unswitching
- Scalarization
- Short circuit elimination
- Strength reduction
- Structure optimization
- Value propagation

All of these are performed by default. The following optimizations are also performed with *–Oscalar3*:

- Conversion of more conditional code to inline select operations
- Aggressive safe value analysis
- Improved short circuit elimination
- More aggressive instances of the default optimizations

When the scalar optimization level is raised to *scalar3*, the optimizations become much more aggressive. Code duplication is more likely to occur, such as with loop unswitching; and extensive compilation times can occur, such as with aggressive safe value analysis and short circuit elimination.

It is safe to specify *scalar3*, and language rules are observed. However, you may see result differences from those obtained with default optimization levels, due to floating point reassociation differences.

In terms of performance, *scalar3* typically improves performance ranging from no gain to 15% or more. It is most useful for codes that contain a lot of conditional code. It is safe, but it can considerably slow your compilation time. If compilation time is not a factor for your application, then Cray recommends that you use *scalar3*.

2.3. -Ovector3; -hvector3

Automatic vectorization is controlled by the *vector* option. By default, it is fairly aggressive; however, if the level is increased to *vector3*, it looks for additional opportunities.

For example, at *vector3* more detailed (and time-consuming) dependency analysis is performed as the compiler tries to find more parallelism. This includes complex dependency testing and more aggressive array privatization analysis.

Another example is that at *vector3*, the compiler more aggressively generates 'safe' code to allow speculative loading of unsafe memory references; typically, these are memory references that may generate a trap if removed from their original context. When this rewrite is performed, it generally allows the elimination of some branching code that is generated by the vectorization process.

The use of *vector3* allows for more and improved vectorization in some codes, the most notable being for C and C++, due to their inherent ambiguities. It is safe to use *vector3*, as it does not unreasonably impact compilation time, and it can significantly improve performance. Cray encourages its use, whether directly or though the *–O3* option.

2.4. -Ostream3; -hstream3

Automatic multistreaming is controlled by the *stream* option. By default, automatic multistreaming is extremely aggressive, and the difference between default optimization and *stream3* is minimal. Primarily, *stream3* allows more aggressive multistreaming of bit matrix multiply (BMM) operations.

The use of *stream3* does not generally yield better application performance. The default multistreaming level is sufficient, as there really is not much else to do at a higher level of streaming. If your code does not use BMM operations, the use of *stream3*, while not beneficial, has no negative impact.

2.5. -Ofp3; -hfp3

The *fp3* option gives the compiler more freedom to optimize floating-point operations. Language standards such as the IEEE floating point standard place severe restrictions on how floating-point expressions can be optimized. Under the default (that is, *fp2*), the compiler is allowed some leeway where performance is critical, so a default compilation is not strictly IEEE compliant. Optimizations such as rewriting floating-point divisions into multiplications by a reciprocal and reassociation of floating-point operands are supported.

However, when *fp3* is specified, the compiler is permitted to perform significantly more aggressive floating-point optimizations. At *fp3*, the compiler is permitted to perform the following optimizations and make the following additional assumptions:

- Assume floating-point comparisons are safe, and will not trap.
- Generate inline code for natural log, exponentiation, and power functions. These inline versions, although as much as 300% faster than library routines, are not quite as precise. However, for most codes the accuracy is more than sufficient.
- Allow more aggressive rewriting of power functions, where a floating-point value is raised to a constant power.

If your application consists primarily of integer operations, this option will not improve its performance. However, for most floating-point intensive applications, this option should be considered if any of the following are true:

- The application has some tolerance for deviations from the IEEE standard.
- The application uses *exp*, *log*, or ** (*pow* in C and C++) intrinsics.

The performance gained with this option ranges from no gain, to 30% (or more) for codes dominated by *exp, log,* or ** intrinsics in performance-critical areas. Cray encourages the use of *fp3* for most codes.

2.6. –Oipa3; -hipa3

The *ipa* option controls the level of interprocedural optimization. Currently, this consists primarily of inlining (including cross-file inlining) and some other performance enhancements such as tail recursion optimization.

The default interprocedural optimization is *ipa3*, which inlines to a depth of one for Fortran and three for C and C++ (as long as the resulting code contains no calls). This is the optimal level of inlining for most codes and should not be changed without good cause.

The levels associated with the *ipa* option are not reflective of any degree of optimization; in other words, *ipa4* will not necessarily produce faster code, and *ipa5* (which should be avoided) actually can produce slower code. Check the appropriate documentation for a complete description of these levels, but in general, this option can be left at *ipa3*.

2.7. –Oclone1

This is a Fortran option. The *clone1* option duplicates procedures when they are called with constant arguments that are not expanded inline. This allows optimization within the *cloned* procedure to take advantage of the constant values, and can lead to better performing code. The primary drawbacks to this option are the increased compile time and larger code size.

The benefit of this option depends on the application. The default is *clone0*, that is, to not clone at all.

2.8. –Oaggress; -haggress

Some individual procedures or functions can become too large, due either to a huge amount of code or too aggressive inlining. In these cases, the compiler uses a technique called *regioning* to logically break the function into smaller parts. The primary impact of this is to control the amount of resources the compiler requires to optimize the function without in turn affecting application performance.

The *aggress* option tells the compiler to *not* perform this regioning action. It essentially tells the compiler to consider all functions as atomic, unbreakable pieces of code and to use all the compilation time and memory necessary to optimize it.

Most codes are not large enough to require regioning, so this option has no effect on them. On codes that do require regioning, this option may provide a slight performance advantage at the price of a potentially

dramatically longer compile time. When optimizing your application, you can use the *aggress* option initially, and selectively remove it if the compile time penalty becomes too severe.

3. Options That Decrease Performance

3.1. –Ooverindex; -hoverindex

The *overindex* option is designed for use with very old codes that have loop nests that have been collapsed by hand. (Modern optimizing compilers automatically collapse loop nests.) Although C and C++ have rules that are somewhat relaxed, it is generally expected that array references will be within the declared boundaries of the array.

When a loop is collapsed by hand, illegal code is introduced into an application – at least one dimension of a multidimensional array will be indexed with a value that is outside its declared range (*overindexed*). The compiler optimizes based on the constraints of the language standards, so if a code violates those standards, incorrect optimization can result. Although C and C++ have rules that are somewhat relaxed when compared to Fortran, it is generally expected that array references will be within the declared boundaries of the array.

The *overindex* option allows you to compile these old codes at the cost of lost optimization. If this option is specified, the primary effect is that range analysis cannot be used to determine the maximum legal trip count for loops. This in turn can lead to poor selection of loops to optimize, less accurate dependence information, and most importantly, the introduction of unnecessary control flow and loop overhead.

A case in point: the *Perfect* benchmark suite that is used at Cray contains 13 codes, with 4,613 vectorizable loops. If *overindex* is specified, 1,243 (or 27%) will run significantly slower, due to additional loop overhead and branch logic. Clearly, this option is undesirable and should not be used.

If it has been empirically determined that an application requires this option to run correctly, it almost certainly illegally over-indexes an array. The best solution in this case is to correct the source code to comply with language standards.

3.2. -Ofp0, -Ofp1; -hfp0, -hfp1

In the same manner that using *fp3* can increase the performance of floating point intensive applications, decreasing the level of this option can have a negative impact on overall application performance.

Unless the application is intolerant of minor floating point variations between hardware platforms, including variations allowed by the language standards, these options should not be used.

3.3. -Oipa5; -hipa5

This option states to inline everything, everywhere, to an unbounded call chain depth, regardless of the size of the resulting code.

Not only can this option lead to unacceptable compile times, it can result in a significant *decrease* in performance.

Forget you ever heard of this option. It is used for in-house stress testing of the compiler. It should never be used for performance enhancement.

-Oipa5/-hipa5 should never be used, period.

3.4. –eh

This is a Fortran option. By default, the compiler converts 8- and 16-bit integer variables to 32-bit integer variables for performance reasons.

The Cray X1 hardware provides support for 32- and 64-bit variables. The implementation of 8- and 16-bit data types is achieved through software. Because of this, these small data types are considerably slower than their 32- and 64-bit counterparts. If your code contains explicit 8- or 16-bit integer variables but does not absolutely depend on 8- and 16-bit storage for correct results, do not use this option.

3.5. –e0; -hzero

These options initialize stack variables to zero at execution time, every time a procedure or function is called. None of the supported Cray language standards require this, and it is primarily useful as a debugging tool.

These options should not be used except to debug an application.

3.6. –ev

This is a Fortran option. It states to place all stack-based user variables in static storage, as if a *save* attribute were placed on the variable declaration. This prevents the compiler from performing some optimizations, such as the best possible dead code elimination, last value capture optimizations, and so on.

Some very old codes may require this option if they have not been updated in the last twenty years or so, as some very old hardware did not support stacks. If you have one of these applications, it should be updated to allow stack storage, and this option will no longer be necessary.

Although the compiler recognizes this option and does what it can to prevent it from degrading performance too much, the option still decreases performance and should be avoided.

3.7. –Oshortcircuit2

This is a Fortran option. It tells the compiler to evaluate logical *and* and *or* operations much how C processes && and || operations, that is, avoiding the evaluation of the second expression if the first is true or false, respectively.

At first glance, this is a good idea. However, it introduces additional tests and jumps which compromise optimization. This is especially true with vectorized code, where the additional conditional logic can dramatically slow down a loop.

The compiler actually has optimizations that try to *reverse* this short circuiting when proved safe; it will even rewrite expressions to make them safe, in order to remove the conditional code. So, it does not make sense to turn on an option to generate slower code, and which the compiler will try to undo.

The default is to short circuit only function calls, which are expensive.

3.8. Other Options

There are other options that decrease performance, but they are uncommon enough they are simply listed here, without going into detail. If you use these options, you should carefully evaluate whether they are actually necessary.

Fortran	C and C++
	-htolerant
	-hnointrinsics
-eL	
-ew	
-Ofusion0, -Ofusion1	***-hnofusion***
-Onoinfinitevl	***-hnoinfinitevl***
-Onointerchange	***-hnointerchange***
-Onorecurrence	***-hnoreduction***
-Onovsearch	***-hnovsearch***
-Ounroll0	***-hnounroll***
-Ozeroinc	***-hzeroinc***

4. Options That Void Your Warranty

4.1. –hivdep

This C and C++ option creates an *ivdep* compiler directive on every loop in the source. It is a holdover from an earlier compiler, can lead to incorrect results, and may actually decrease performance by limiting parallelization to innermost loops.

It should not be used under any circumstances.

5. Providing More Information to the Compiler

5.1. –Ossp; –hssp

This option instructs the compiler to not perform automatic multistreaming, and to produce code for a single-streaming processor (*ssp*) of a multistreaming processor (*msp*).

On first examination, this may appear to be an option that is guaranteed to reduce performance. However, if carefully used, it can increase performance *from an overall throughput standpoint*, even if it decreases performance on the individual application.

When compiling for multistreaming (the default), if any part of an application is multistreamed, then the entire executable is tagged as a multistream application. Normally, this is the desired result. However, if an application has little stream-level parallelism, as can happen with Gnu utilities and similar codes, then it is better to disable multistreaming altogether.

By compiling an entire application with the *ssp* option, no multistreaming will be performed, and the resulting executable will run on only one processor of a multistreaming processor. This allows for four concurrent copies of the application (or other single-streamed executables) to be run.

Compare this to a version of the application that is only slightly multistreamed: when the multistreamed portion of the application is executed, all four processors are in use. Conversely, when the single-streamed portion of that same code is executed, only one processor is in use, and the other three stand idle. But if you use *–hssp*

on the same code, one processor is used and the other three are free to run other SSP jobs.

In general, most applications should not run in *ssp* mode. If there is any doubt, the application can be built in both *msp* and *ssp* modes, and the performances compared to determine whether the *ssp* option should be used.

5.2. –hrestrict=f

This C and C++ option marks pointers which are function parameters with the *restrict* attribute. This gives C and C++ pointers roughly the same aliasing attributes that Fortran dummy arguments have.

Unlike *–hivdep*, which should never be used, this option can be useful for well-behaved codes. Many structured codes use pointers in a very restrained and compiler-friendly manner, which allows for the use of this option. The performance benefit from this option can range from no gain to huge gains thanks to additional vectorization and multistreaming. However, if the *–hrestrict=f* option is used improperly, incorrect results can occur. It is up to the developer to decide when it is legal to use.

5.3. –Onopattern; -hnopattern

By default, the compiler looks for code constructs that can map onto selected *libsci* routines, which have traditionally mapped to common idioms such as matrix multiplications. In the past, these hand-optimized *libsci* routines were substantially faster than compiler-generated code. With compiler optimization improvements over the years, the compiler now typically produces code that is as good or actually exceeds the performance of the library routines. Accordingly, the number of patterns the compiler matches is now quite small. What it does match, however, are constructs that are still faster if the *libsci* version is used for a typical application.

Although generally not recommended, the *nopattern* option can tell the compiler to generate inline code for everything. The applicability of this directive can be determined if you compare builds and runs with and without it.

6. Recommended Command Line Options for Performance

6.1. The Command Line

All the options that decrease performance should be avoided. For the simplest command line that has the greatest impact, the following is recommended:

Fortran:
-O3 –Ofp3

C and C++:
-O3 –hfp3

The use of the *O3* is an easy way to specify *scalar3*, *vector3*, and *stream3*. Although *stream3* does not provide much of a performance boost, *scalar3* and *vector3* can result in significant performance enhancements. These options are generally safe to use as well.

The *fp3* option is useful for codes with significant floating-point operations. It can lead to slightly different results than the default *fp2*, due to differences in reassociation, or in the case of the inlined math functions, different algorithms. This difference is acceptable for most applications.

If the application is intolerant of slight floating-point differences, you may need to remove the *fp3* option. If the compilation time is unacceptable, you may need to remove the *O3* option.

6.2. Why These Options Are Not Default

The recommended options are not performed at default for different reasons. The *O3* option is not the default because of the additional compilation time that can it can incur, and because of the potential for larger executables due to code duplication. The *fp3* option is not the default because the potential for floating point differences, although generally acceptable, is simply too aggressive to apply automatically.

6.3. Why Not the aggress O*ption?*

Some application developers use other optimization flags on a regular basis. Options that decrease

performance significantly are specified above, but one flag that is commonly used is the *aggress* flag.

The use of *aggress* is not generally recommended, as the potential for improved performance is slim, but the potential for dramatically increased compilation time is large. In terms of a cost/benefit analysis, the return is poor.

6.4. Compilation Time

The primary contributors to the overall compilation time are inlining and multistreaming, controlled by the *ipa* and *stream* options, respectively. Although it is not recommended to turn either option off, if compilation time is a problem, consider first compiling with *ipa0* to disable inlining. If it is still a problem, add *stream0*. Both of these options will create overall performance degradations for typical code, however, so these actions are not recommended.

7. Conclusion

The default optimization level provided by the Cray X1 compilers exploits multiple levels of parallelism, including SIMD and MIMD parallelism. Although this default level of optimization provides very good performance for most codes, this performance can potentially be improved by using additional compiler flags.

Conversely, some compiler flags can cause significant performance degradation and should be avoided. A general recommendation for compiler options for maximum performance is:

Fortran:
-O3 –Ofp3

C and C++:
-O3 –hfp3

This combination strikes a balance between maximum performance and application safety.

Acknowledgements

Thanks to all who reviewed this document, and who provided information for it.

Reference Materials

Consult the following Cray reference manuals for further information:

- S-2315-54: Optimizing Applications on Cray X1 Series Systems
- S-3901-54: Cray Fortran Compiler Commands and Directives Reference Manual
- S-2179-54: Cray C and C++ Reference Manual

About the Author

Terry Greyzck is a graduate of Michigan State University, and has worked for Cray Inc. for 20 years. He started by providing field support at the Livermore Department of Energy laboratories, and eventually migrated to compiler optimization work, with an occasional management stint thrown in for good measure.

Terry's first vector translator was written for the Cray Ada compiler. From there, the next step was to develop vector translation, multistreaming technology, and late-stage optimizations for the current Cray X1 common optimizer, which is used for Fortran, C, and C++.

Terry currently works out of the Cray Inc. office in Mendota Heights, Minnesota. He can be reached at:

Terry Greyzck, Cray Inc.
1340 Mendota Heights Road
Mendota Heights, MN 55120
(651) 605-8979
tdg@cray.com