Computers in Physics

C++ Gets Faster for Scientific Computing

Arch D. Robison and Paul F. Dubois

Citation: Computers in Physics **10**, 458 (1996); doi: 10.1063/1.4822473
View online: http://dx.doi.org/10.1063/1.4822473
View Table of Contents: http://scitation.aip.org/content/aip/journal/cip/10/5?ver=pdfcov
Published by the AIP Publishing

C++ GETS FASTER FOR SCIENTIFIC COMPUTING

Arch D. Robison

Department Editor:
Paul F. Dubois
dubois1@llnl.gov

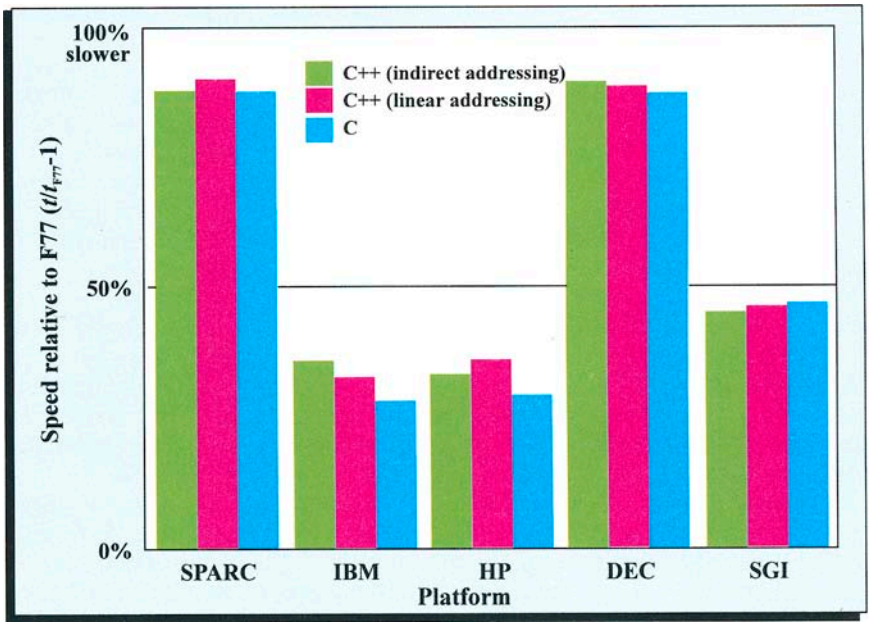

Figure 1. Speed comparison between C++ and C code and Fortran (F77) code for Haney's benchmark that multiplies 100 × 100 real matrices. The C++ and C codes were compiled by KCC; the Fortran code by the vendor's Fortran compiler. Two implementations of the C++ matrix classes are timed: one that accesses data in a linear manner and the other that uses indirect addressing. This is a big improvement over earlier C++ compilers. In Haney's study the C++ speeds were much slower relative to Fortran: about 300% for indirect addressing and 400% for linear addressing.

C++ offers some substantial improvements over C and Fortran for scientific programming. Two particular C++ features that assist scientific programming are classes for encapsulating abstractions and operator overloading for expressing formulae. However, use of these features can incur a large performance penalty that makes C++ unsuitable for compute-intensive tasks.

Progress in programming techniques and compilers is shrinking the performance penalty. In some cases there is no longer a penalty. This article discusses some of the performance issues, new programming techniques, and a new compiler developed by the author and his coworkers at Kuck & Associates Inc. (KAI)[1] that solves some major performance problems.

Style versus speed

The author became acutely aware of the C++ performance problem while working on high-performance codes for seismic imaging at Shell Development. These codes consumed CPU-months on supercomputers; thus performance was paramount. Depending upon how C++ was used, its performance was acceptable or unacceptable.

The problem was not that C++ is intrinsically slow compared to C. Indeed, a C program compiled with a C++ compiler will yield about the same performance. The problem is that C++ style tends to be quite different from C style. C programs tend to emphasize computations on scalars. Thus implementors of C have focused on optimizing scalar computations. For instance, allocating a variable in a machine register is often preferable to allocating it in memory, because accesses to registers are quicker. All modern compilers try to allocate scalar variables to registers when profitable. However, many C compilers refuse to store nonscalar variables in registers. Although it would be technically possible for the compilers to do this, few existing C programs would benefit. In contrast, C++ programs emphasize computations on objects, not scalars. Consequently, C optimizers fare poorly on C++ style.

Perhaps the most striking demonstration of poor optimization occurs when numerical programmers introduce classes for complex numbers. Compared to code using "longhand" scalar arithmetic for the same computation, classes for complex numbers typically incur a two- to four-fold penalty in speed.

C++ style also tends to use more pointers than C style. The primary reason is that run-time polymorphism in C++ depends heavily on using pointers or references. Since there is little implicit in C programs, C programmers tend to be conscious of their use of pointers. Often, C programmers optimize code manually by loading frequently used pointers into temporary variables. C++ programs are harder to hand-optimize. For example, from a performance viewpoint, C++ references are simply pointers that are assigned exactly once. The indirections are implicit, and the programmer becomes less conscious of them. Moreover, the reference or pointer may be a private part of an abstraction and not accessible for the programmer to optimize by hand.

Most C++ compilers get their optimizers as hand-me-

Arch D. Robison is the lead developer for KCC at Kuck & Associates Inc., 1906 Fox Drive, Champaign, IL 61821. E-mail: robison@kai.com

downs from C. The consequent impedance mismatch between style and optimizer leads to poor performance. A remedy adopted by many (including the author while at Shell) is to toss the benefits of C++ and write the performance-critical parts of code in C style. This solution of "hand-lowering" code is not unique to C++. Certainly it is a rare Fortran programmer who has not at one time or another called library routines written in assembly code.

State of the art in commercial C++ compilers

At KAI we examined the foundations of classical optimizations for Fortran and rebuilt them to work on C++ programs. The resulting KAI C++ compiler (KCC)[1] is thus designed from the start to optimize C++. The KCC optimizations are not based on recognizing certain classes as special. For example, Fortran compilers can easily optimize calculations involving complex numbers because they are built into Fortran. It was tempting to build an optimizer that recognizes a special class for complex numbers and exploits ad hoc knowledge about complex numbers. Such a type-specific approach, however, goes against the spirit of C++. C++ lets users extend the language by defining new types and overloading operators for those types. A type-specific optimization would not apply to those new types. Thus the KCC optimizer relies on general optimization principles and not on recognizing certain classes. Unlike Fortran optimizers, it is equally adept at optimizing complex numbers as classes for other small objects such as quarternions and three-space vectors.

Haney's benchmarks

On these pages, Scott Haney of Lawrence Livermore National Laboratory questioned the suitability of C++ for high-performance computing.[2] He presented comparative performance benchmarks that he obtained by running C++, C, and Fortran 77 versions of three kernels. The kernels were matrix multiplication of real matrices, matrix multiplication of complex matrices, and a vector expression.

This article makes use of Haney's benchmarks. Although others are available,[3,4] Haney's are more appropriate for the present discussion because:

- They have been published and are relevant to scientific computing.
- They illuminate a range of performance problems that have been solved, at least relative to C.
- They compare C++ with C and Fortran 77.

The table on p. 461 summarizes equipment and software options used in the benchmarking tests.

Having a Fortran baseline is particularly important. In debates about the relative speed of C++ and C, participants sometimes forget about Fortran 77. Fortran 77 is an acknow-

Feedback

Scott Haney is preparing an article to complement this month's article on C++ optimization. It will cover the recent development called "expression templates." Scott says he has added a lot of goodies to the arrays, such as:

- making them "self-counting";
- adding array sections;
- adding boolean expressions; and
- adding a where statement.

The Fourth International Python Conference was held here at Lawrence Livermore National Laboratory (LLNL) in June, and developments in the Python world continue to excite me. You can read all about it on the Python Web pages (http://www.python.org), and of course I hope you enjoyed our Python series earlier this year.

Speaking of learning, I've learned that writing a book is a lot harder than it looks. I just finished *Object Technology for Scientific Programming*, which should appear about the same time as this article does, from Prentice-Hall. I wrote it at home, not as part of my job at LLNL. I used Framemaker on a PC, which I can recommend as very good for this sort of large effort. Framemaker has a book feature that allows you to divide the book into separate files for each chapter. The automatic generation of indices, table of contents, full control of page numbering, layout, and so on, all add up to a really professional desktop-publishing package. It lacks a book-wide search and replace. A product available from Quadralay (info@quadralay.com, http://www.quadralay.com) translates your Framemaker document into Hypertext Markup Language; a simple version is built into Frame 5. You can move the Frame files back and forth between your PC and Unix boxes, and to and from a variety of other word-processing formats.

Jeff Templon of MIT, author of the article about Linux in the January/February 1996 issue, dropped me a note about recent developments. He says, "I made quite heavy mention of Microway and their then-soon-to-be-released Fortran compiler. They never really delivered—the compiler does run under Linux but is not a professional product. I'd like to let the people who read your column know this." Jeff's Fortran page is located at http://marie.mit.edu/~templon/fortran.html; on it he has further discussion on this issue.

I continue to hear marvelous stories of managers rushing in demanding that the software people do a project in Java. The managers haven't a clue what Java is, of course. Sometimes I expect to meet Dilbert in person at the cafeteria. I went to "Java Camp" for a day so I'm practically an expert. Unless someone out there who has been to Java Camp for two days offers to write an article, I may end up doing one myself. My first impression is that Java is C++ rewritten by someone who liked Eiffel, but unfortunately it lacks support for assertions. Don't expect it to be fast, but for the right project it might make sense to learn it.

Paul F. Dubois

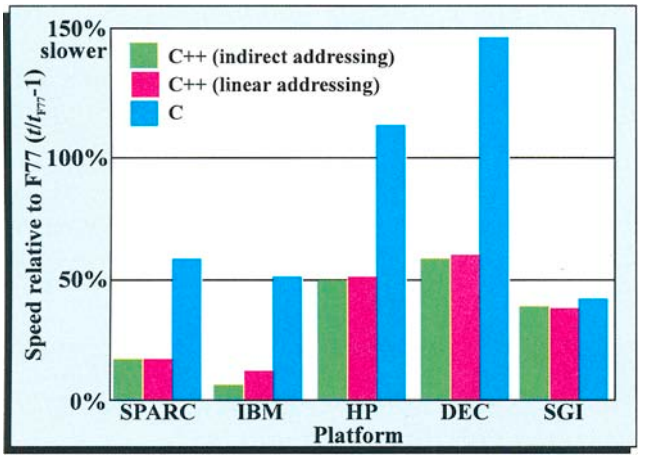

Figure 2. Speed comparison between C++ and C code and Fortran (F77) code for Haney's benchmark that multiplies 100×100 complex matrices. In Haney's original study, the C++ codes ran at least 200% slower than Fortran. A striking feature here is that with KCC, the C++ code runs faster than the C code on most of the platforms, because the use of objects improves locality of reference. Furthermore, KCC optimizes the linear-addressing implementation so well that the extra complexity of the indirect-addressing implementation is hardly worth the effort.

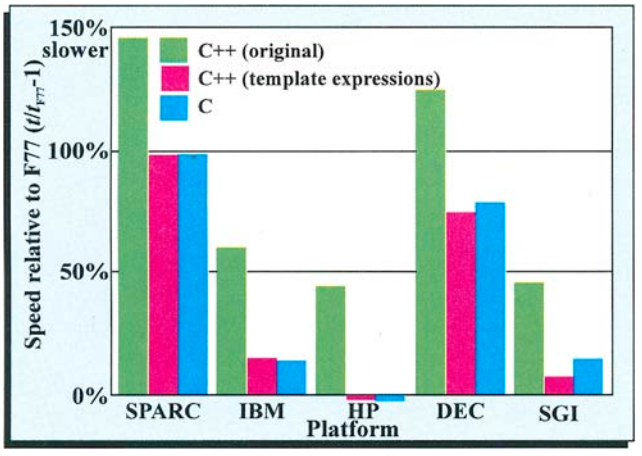

Figure 3. Speed comparison between C++ and C code and Fortran (F77) code for Haney's vector inductance calculation on several platforms. Two implementations of the C++ vector classes are timed: one using Haney's original "optimized" classes (green bars) and the other using Haney's new template-expression classes. The combination of template expressions and KCC delivers performance equivalent to hand-written C code.

ledged performance leader in scientific programming. (Its successor does not enjoy such a favorable reputation. Some wags suggest that Fortran 90 is a step toward making Fortran run as slowly as C++.)

Haney's benchmarks bring to light several inefficiencies of code optimizers for C++ as compared to C optimizers. They also showcase widely known inefficiences of C++ and C compared to Fortran. These inefficiencies arise because Fortran's conservative prohibitions against aliasing are absent from C++ and C. (Two identifiers are aliases if they refer to the same memory location.) This article focuses on the C++ versus C issue, where the differences arise from optimizers and not language semantics. The next three sections parallel three sections of Haney's article.

Minimal use of C++ features

The first Haney kernel measures the time to multiply single-precision 100 × 100 real matrices. The inner loops of such a calculation contain opportunities for two classical optimizations: invariant hoisting and induction variables. Invariant hoisting "factors out" invariant computations from the inside of a loop to outside the loop. Induction variables reduce expensive multiplications with additions by exploiting difference equations. For example, consider the following code fragment.

```
for(int i=0; i<N; i++)
  a[m*i] = x/y;
```

A typical C optimizer might compile the code as though it were written:

```
if (N>0) {
  p = &a[0];
  t = x/y;          /* Compute invariant */
  do {
    *p = t;
    p += m;         /* Bump pointer by difference */
  } while(++i<N);
}
```

Modern C and Fortran compilers are good enough at this sort of optimization for scalars that programmers need not be bothered doing the optimization by hand. In fact, there usually are invariants and multiplications that are hidden in the C and awkward for the programmer to optimize. In the example above, for instance, there is a hidden multiplication in p+=m; at the machine level it becomes the equivalent of (char*&)p += sizeof(*p)*m.

In practical C and Fortran programs, the invariants and inductions concern scalars. But in C++, they often concern members of objects too. Optimizers that do not peer into objects miss the opportunities.

Haney tested two implementations of his matrix classes. The linear-addressing implementation mapped matrices onto storage with a linear mapping similar to that employed by Fortran and C matrices. The indirect-addressing implementation used an array of pointers to rows. Haney found that the latter, though more complicated, ran faster than the former because the C++ compilers he tested failed to exploit invariants and induction variables.

For the compilers tested by Haney, the C++ version typically ran twice as slowly as the C version. With KCC, however, the results are much better. As Fig. 1 shows, this compiler eliminates the penalty relative to C. Furthermore, KCC delivers C speed from the C++ code with linear addressing. The extra complexity of adding indirect addressing offers no significant improvement. Indeed, if the inner loops of the matrix multiplications addressed the matrices along columns instead of rows, the extra indirection would cause loss of performance. This is a nice example of a general point: The purpose of an optimizer is not only to make a machine run faster, but also to let programmers write code more simply.

Poor optimizers lead programmers to convolute programs for sake of efficiency. Good optimizers let convolution remain an algorithm, not a programming practice.

Complex arithmetic

The second Haney kernel measures the time to multiply single-precision 100×100 complex matrices. Thus this kernel adds a new complexity for which C optimizers are ill-prepared: temporary objects. Classical optimizers are excellent at optimizing temporary scalar values that arise from traditional arithmetic expressions. The temporaries are put into registers rather than memory or, if constant, perhaps optimized away. But use of operator overloading introduces new kinds of arithmetic and temporaries. For example, Haney's benchmark defines arithmetic on complex numbers. Optimizers used to dealing with only scalar temporaries fail to put the complex temporaries into registers.

Figure 2 shows that the KCC optimizer practically eliminates the penalty relative to C. Often, in fact, the C++ version is faster than the C version.

How can the C++ version be faster? Pushing a machine to its limit can amplify the effects of code-generation "noise." These "noise" effects include data and instruction alignment and cache effects. However, the large differences here are more than noise. They are a fortuitous consequence of programming with objects instead of scalars. The inner core of the C version handles the real and scalar parts separately, and looks like this:

```
CTC[i - 1 + M * (j - 1)].re +=
   rtemp * CAC[i - 1 + M * (k - 1)].re -
   itemp * CAC[i - 1 + M * (k - 1)].im;
CTC[i - 1 + M * (j - 1)].im +=
   rtemp * CAC[i - 1 + M * (k - 1)].im +
   itemp * CAC[i - 1 + M * (k - 1)].re;
```

The inner core of the C++ version uses overloaded arithmetic on complex numbers, and looks like this:

```
CT(i,j) += temp * CA(i, k);
```

Besides the notational convenience, there is a subtle difference in the ordering of reads and writes to memory. The C version updates the real part and then updates the imaginary part. The C++ version computes the entire right-hand side and then updates the left-hand side. (The C version could be changed to do this too.) In general, making references to adjacent memory locations adjacent in execution helps. Unfortunately, subtle interactions of register allocation and cache organization make the situation complicated in most cases, and so there is no guaranteed rule for exploiting this principle. Programmers must resort to stochastic optimization (colloquially known as "random tweaking").

Vector operations

The third Haney kernel, which deals with vector expressions, demonstrates a quite different inefficiency. The C++ version looks like this:

```
li = Mu0 * R *  (0.5*(1.0 + (1.0 / 24.0) * sqr(w / R)) *
   log(32.0 * sqr(R / w)) + 0.05 * sqr(w / R) - 0.85);
```

The variables R and w are vectors, and the various operators are overloaded to operate on vectors. Each vector operator is implemented as a separate loop. The Fortran and C versions use a scalar arithmetic with a single loop.

Here is the C version:

```
for( i=0; i<N; i++ )
   li[i] = Mu0 * R[i] * (0.5 * (1.0 + (1.0 / 24.0) *    sqr(w[i] / R[i])) *
      log(32.0 * sqr(R[i] / w[i])) + 0.05 * sqr(w[i] / R[i]) - 0.85);
```

The C++ version suffers from the having to allocate, write, read, and deallocate the vector temporaries for each vector operation. This example illustrates an important point: No matter how efficiently individual operations might be written, their composition may be relatively inefficient. This point applies just as well to assembly-code libraries for Fortran. The time to execute two library calls is at best the sum of the times for the individual calls.

The linearity can be beaten only by optimization that alloys multiple operations. In the case of the third Haney benchmark, this requires that an optimizer transform the C++ version into the C version by fusing loops. Though loop fusion is a common optimization in Fortran, the liberal aliasing rules of C and C++ put it beyond the state of the art for C and C++ in practical programs. Fortunately, there is a way to rewrite the vector libraries so that they do the loop fusion themselves. Todd Veldhuizen's new technique, called template expressions, exploits the C++ template mechanism to perform compile-time transformations of code.[5] Expression templates apply the notion of template metaprograms, which use the C++ template mechanism to carry out arbitrary computations during compilation.

Template expressions alone do not cure all performance problems. They make heavy use of objects and thus cannot make up for an optimizer's poor handling of objects. However, in conjunction with a good object optimizer, they become a powerful technique that lets users specify domain-specific transforms.

Scott Haney rewrote the C++ version to use template expressions.[6] The C++ version of the test looks the same, except that the vector classes are implemented using template expressions. Figure 3 shows the speed delivered by Haney's original vector classes, the new template-expression classes, and C code when compiled by KCC. The timings are for vectors of length

Table. Platforms, compilers, and options used for comparisons.

Platform	C++		Fortran	
	Compiler	Options	Compiler	Options
Sun SparcStation 10/41	KCC	+K3 -fast -O4	f77 (SC4.0)	-fast -O4
IBM RS6000 model 580	KCC	+K3 -O3	xlf (v3.02)	-O3
HP 9000/735	KCC	+K3 -O2	f77 (beta 10.01)	-O3
DEC Alpha 3000/500	KCC	+K3 -O4 -migrate	f77 (v3.6)	-O5 -fast -migrate
SGI R4000	KCC	+K3 -O2	f77 (Irix 5.3)	-sopt -O2

100. With template expressions and KCC, the abstraction penalty is now practically zero. (In fact, other code-generation and measurement noise makes it appear slightly negative in some cases.)

The vector problem is mostly, but not completely, solved. Haney reports a 30% penalty for short vectors (length = 3), and that some inefficiencies are hidden by the relatively complicated expressions in his kernel.[6] However, combining a different template-expression technique with some new optimizations in the next-generation KCC optimizer may solve the short-vector problem.

Style *and* speed

Matching Fortran's speed remains an unsolved problem. For some problem domains, clever use of template expressions can allow C++ libraries to exploit domain-specific algebraic laws, resulting in codes that run faster than Fortran. However, for simple array crunching, Fortran is likely always to be king due to its aliasing restrictions. In theory, extensions such as the Numerical C Extensions Group's proposed keyword restrict give C and C++ the same power as Fortran, but only if the programmer writes Fortran-style code.

It cannot yet be said that C++ is always fast enough for scientific computing. Other features of C++, such as virtual functions and exceptions, still incur performance penalties. However, with modern optimizers designed for C++, such as KCC's, it is much faster than it used to be. The combination of an optimizer designed for C++ programming style coupled with template expressions removes the C++ abstract penalty (relative to C) from all three of Haney's benchmarks.

An optimizer designed for objects not only makes the machine run faster—it opens up programming in higher-level styles while retaining the performance of C. In particular, programs can be written at the level of overloaded operators and small objects such as complex numbers and three-space vectors, rather than lowered to a soup of simple numeric variables. Perhaps high-performance scientific codes can now have less code and more science.

Acknowledgments

Scott Haney of Lawrence Livermore National Laboratory and Dan Quinlan of Los Alamos National Laboratory contributed benchmarks to us that influenced the theory behind the KCC optimizer.

References

1. KAI, KAI C++, and KCC are trademarks of Kuck & Associates Inc., 1906 Fox Drive, Champaign, IL 61821.
2. Scott Haney, Comput. Phys. **8**, 690-694 (1994).
3. Kent G. Budge, James S. Peery, Allen C. Robinson, and Michael K. Wong, Journal of C Language Translation **6** (1994).
4. Arch D. Robison, "OOPACK benchmark," posted to the Usenet newsgroup comp.lang.c++ in October 1993. Copy available from http://www.kai.com/oopack/oopack.html.
5. Todd Veldhuizen, C++ Report, June 1995, p. 26–31. For current information, see http://monet.uwaterloo.ca/blitz.
6. Scott Haney, personal communication.

See CIP Interactive Reader Service, p. 504