Will C++ be faster than Fortran?

Todd L. Veldhuizen and M. Ed Jernigan

Department of Systems Design Engineering, University of Waterloo,
Waterloo Ontario N2L 3G1 Canada
tveldhui@monet.uwaterloo.ca

Abstract. After years of being dismissed as too slow for scientific computing, C++ has caught up with Fortran and appears ready to give it stiff competition. We survey the reasons for the historically poor performance of C++ (pairwise expression evaluation, the abstraction penalty, aliasing ambiguities) and explain how these problems have been resolved. C++ can be faster than Fortran for some applications, due to template techniques (such as expression templates and template metaprograms) which permit optimizations beyond the ability of current Fortran compilers.

1 Introduction

As of 1994, the performance of C++ for scientific computing was disappointing. Typical benchmarks showed C++ lagging behind Fortran's performance by 20% to a factor of ten. Performance was so poor that mixed-language programming was necessary: a program's framework could be written in C++, but any speed-critical routines had to be coded in Fortran.

In the past five years there have been significant developments. There are now C++ compilers which perform C++ specific optimizations, such as KAI C++ (from Kuck and Associates Inc.) and Intel C++. Two new programming techniques (expression templates and template metaprograms) perform optimizations such as loop fusion and algorithm specialization. Recent benchmarks show C++ encroaching steadily on Fortran's high-performance monopoly. The latest results are so encouraging that we can pose a heretical question: Will C++ be *faster* than Fortran?

We'll start by looking at reasons why C++ used to be slower, and then describe techniques which can make it faster than Fortran. Ideas will be illustrated with examples and benchmarks from the Blitz++ class library.

2 Why was C++ slower?

2.1 Pairwise expression evaluation

Operator overloading in C++ permits a natural notation for array operations. For example, one can write `z = w + x + y;` where w, x, y, and z are vectors. Unfortunately, overloaded operators in C++ are always evaluated in a pairwise manner. In a straightforward implementation, evaluation of `z = w + x + y;` will result in two temporary vectors and three loops being generated. For large vectors, speeds of about 40% Fortran can be achieved (depending on the expression). For tiny vectors ($N < 10$), the cost of allocating temporaries dominates, and the speed is 5-10% Fortran speed.

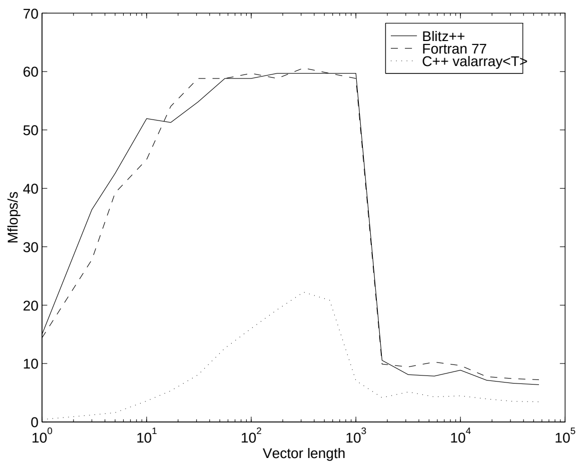

Fig. 1. Results for the DAXPY benchmark

This problem has been solved by the expression templates technique [7]. Expression templates are used to build parse trees of expressions at compile time; the parse trees are encoded as type names. Once the parse tree has been encoded, it can be manipulated in interesting ways. One application of expression templates solves the pairwise expression evaluation problem: code such as `z = w + x + y;` is transformed into

```
for (int i=0; i < N; ++i)
    z[i] = w[i] + x[i] + y[i];
```

Expression templates have some overhead associated with constructing the parse tree. However, KAI C++ optimizes away this overhead completely, resulting in code which is just as fast as Fortran 77.

Figure 1 shows benchmark results for a DAXPY operation implemented using the Blitz++ library, Fortran 77 and the `valarray<T>` class from the ISO/ANSI C++ library.[1] The `valarray<T>` performance is typical of a pre-expression tem-

[1] All benchmark results were measured on a 100 MHz RS/6000 Model 43P using KAI C++ at +K3 -O3, XL Fortran 77 at -O3, and XL Fortran 90 at -O3. The XL Fortran 90 compiler was omitted from the DAXPY benchmark because it performs poorly. DAXPY is the Level 1 BLAS which implements the vector operation `y += a * x`.

plates implementation. The sharp performance drop around N=2000 is due to cache effects.

Although Fortran 90 supports operator overloading, there's no comparable mechanism to expression templates. This limits the generation of efficient kernels by F90 compilers to expressions involving arrays which are dense, asymmetric, box-shaped, and linearly-addressable. The expression templates mechanism in C++ has no such limitations: it can generate efficient kernels for sparse, symmetric, jagged-edged, and/or nonlinearly-addressed arrays.

2.2 Abstraction Penalty

It's sometimes remarked that there's no reason C++ should be inherently slower than Fortran, since you can write the same low-level code in both languages. Taking this argument to extremes, one could argue that C++ is inherently *faster* because it supports the `asm` { ...} construct! But the promise of C++ is flexible, maintainable, abstract code. In the past, abstraction in C++ has been accompanied by a performance loss often referred to as the "abstraction penalty". The abstraction penalty was caused by the inability of compilers to do C++ specific optimizations; more recent compilers such as KAI C++ reduce this penalty substantially [5]. Experience with Blitz++ has shown that performance can be tweaked to match Fortran 77/90.

2.3 Aliasing

C++ compilers must assume that two pointers might refer to the same data, preventing certain optimizations from taking place. Fortran compilers don't have this problem, since the language doesn't permit aliasing ambiguities.

The Numerical C Extensions Group (NCEG) introduced a pointer qualifier `restrict` to advise the compiler that no aliasing is possible. A proposal to add this keyword to the ISO/ANSI C++ standard was rejected, but some C++ compilers support it anyway (KAI C++, Cray C++). The C++ standard does state that compilers can assume no aliasing when dealing with the standard array class `valarray<T>`, but this is likely of limited usefulness.

Aliasing is a problem primarily for code which is written at a low level of abstraction, and uses pointers directly. If code is written at a high level of abstraction using a class library, it's generally possible for the library designer to avoid aliasing problems, for example by loading data into local variables which are guaranteed alias-free.

3 Why might C++ be faster?

3.1 Abstraction can make optimization easier

Optimization of low-level languages such as Fortran 77 requires a transformational approach: a compiler must first understand the intent of the program, and then transform it into a faster (but equivalent) program. Optimizing code written at a high level of abstraction is often easier, since a generative (rather than transformational) approach is possible. Abstract programs resemble specifications more than implementations. Complex, machine-specific code can be generated to carry out the operation.

In Fortran 90/95, the array abstraction is built into the language; the compiler generates optimized low-level code to implement the high-level program. C++ has an advantage because it provides library designers with the tools they need to build their own abstractions; optimized code can be generated by the library itself using template techniques.

Consider this code excerpt which implements the 2-D acoustic wave equation using Blitz++ arrays:

```
Array<float,2> P1(N,N), P2(N,N), P3(N,N), c(N,N);   // ...
Range I(1,N-2), J(1,N-2);

for (int iter=0; iter < niters; ++iter)
{
    P3(I,J) = (2-4*c(I,J)) * P2(I,J)
      + c(I,J)*(P2(I-1,J) + P2(I+1,J) + P2(I,J-1) + P2(I,J+1))
      - P1(I,J);

    P1 = P2;
    P2 = P3;
}
```

The performance of this stencil operation is constrained by memory bandwidth. Three optimizations are required for good performance: tiling, partial unrolling, and avoiding the array copy operations. Blitz++ performs the first two optimizations automatically, and the third is achieved by a single line of code. It's possible to do the same optimizations in Fortran 77 and 90, but the clarity of the code suffers.

Untuned versions	Time	Mflops/s	Code statements
Fortran 90	93.3 s	9.7	12
Fortran 77	53.2 s	17.0	21
Blitz++	57.8 s	15.7	9
Older C++ math library	418.2 s	2.1	9
Tuned versions			
Fortran 90	41.1 s	22.1	19
Fortran 77	27.3 s	33.2	39
Blitz++	31.6 s	28.7	8
Older C++ math library	361.9 s	2.4	11

Table 1. 2D Acoustic benchmark results for 240 iterations on a 650^2 grid.

Table 1 shows benchmark results for this problem[2]. The Blitz++ versions are highly expressive (very few lines of code are required to implement the benchmark) and compete with the performance of the Fortran versions. The tuned Blitz++ and Fortran 77 implementations do exactly the same optimizations. The tuned Fortran 90 version avoids array copying, but leaves the traversal order up to the compiler. Although the Fortran 77 version is slightly faster than Blitz++, it took 30 lines of source code and several days of painstaking tuning to achieve this result, and the code is not reusable for other stencils. The Blitz++ library generates an equivalent implementation for any array stencil, triggered by a single line of source code. Moreover, the code which it generates can be platform specific.

It is possible to do better than the tiled traversal, by using an iteration space tiling (IST) (see e.g. [10]). Such tilings are fiendishly complicated to implement, and are currently far beyond the ability of optimizing compilers. In C++, it's possible to create template-based frameworks for ISTs. Such a framework for finite-differenced PDEs would require users to provide their stencil operation and boundary conditions. These would be combined with the generic IST code to produce a near-optimal implementation. It's not possible to implement such frameworks in Fortran 77/90/95, since these languages lack generic programming features.

In addition to generating special cache-optimized traversals for 2D and 3D stencil operations, Blitz++ does other loop transformations which previously had been the sole domain of optimizing compilers, such as loop interchange, hoisting stride calculations out of inner loops, partial unrolling, collapsing inner loops, and optimizations for unit strides. Features "on the drawing board" include automatic padding of arrays to avoid cache interference, and loop fusion for multiple array expressions.

3.2 Directed algorithm specialization

It's often possible to speed up an algorithm by specializing it for a particular situation. Scientific computing algorithms are known to benefit greatly from specialization [1]. Consider this dot product algorithm:

```
double dot(double* a, double* b, int n)
{
    double result = 0.0;
    for (int i=0; i < n; ++i)
        result += a[i] * b[i];

    return result;
}
```

[2] Source code is available as part of the Blitz++ library: see `http://monet.uwaterloo.ca/blitz/`.

If we wanted to use `dot()` for vectors of length 3, the performance would be a fraction of peak performance due to function call and looping overhead. A specialized version eliminates the performance loss:

```
inline double dot3(double* a, double* b)
{
    return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}
```

There are two competing approaches to generating specialized algorithms: partial evaluation and generative metaprogramming. Partial evaluators use a transformational approach to optimization: they take a program, analyze it, and transform selected algorithms into specialized versions. Many compilers perform very limited forms of partial evaluation.

Generative metaprogramming provides an alternative to partial evaluation. The term implies programs which generate source code for specialized algorithms. The technique is well-known among computational scientists: examples include producing source code for fixed-size, unrolled Fast Fourier Transforms and matrix multiplication code for matrices with known sparse structure. It turns out that C++ provides crude generative metaprogramming facilities quite by accident, as a byproduct of its template instantiation mechanism [8]. These *template metaprograms* are capable of doing arbitrary computations at compile time, and produce compilable C++ code as their output.

Template metaprograms can be used to specialize algorithms. For example, an FFT implemented using template metaprograms can calculate the appropriate complex roots of 1 and array shuffle at compile time, unroll all the loops, and output a solid block of floating-point math code. An important distinction between template metaprograms and partial evaluation is that the algorithm specializations produced by template metaprograms are unlimited in complexity and *directed* – one has complete control over exactly which optimizations are performed.

A prime candidate area for algorithm specialization in scientific computing is manipulation of small, dense vectors and matrices [9]. Consider this equation for a reflecting an incident ray $\mathbf{x}$ off a surface with normal $\mathbf{n}$:

$$\mathbf{y} = \mathbf{x} - 2(\mathbf{x} \cdot \mathbf{n})\mathbf{n} \tag{1}$$

A Blitz++ implementation of this equation would use the `TinyVector<T,N>` class:

```
typedef TinyVector<double,3> ray;

void reflect(ray& y, const ray& x,
    const ray& n)
{
    y = x - 2 * dot(x,n) * n;
}
```

The `TinyVector<T,N>` class places the elements of the vector directly on the stack, so there is no memory allocation overhead. Using a combination of template metaprograms and expression templates, the above code is transformed into

```
double _t1 = x[0] * n[0] + x[1] * n[1] + x[2] * n[2];
double _t2 = _t1 + _t1;
y[0] = x[0] - _t2 * n[0];
y[1] = x[1] - _t2 * n[1];
y[2] = x[2] - _t2 * n[2];
```

The back-end compiler then reorders this code for superscalar execution, and produces a mere 20 instructions. Specializing algorithms for small vectors and matrices exposes low-level parallelism, eliminates function call overhead, permits data to be registerized, and allows floating point operations to be scheduled around adjacent computations. The resulting algorithms are up to 10 times faster than calls to library subroutines.

3.3 Encapsulation improves locality of reference

Encapsulation is a basic idea of object-oriented design: you take data which is closely related, and package it as an object. A side benefit of encapsulation is that it encourages locality of reference, resulting in favorable memory access patterns [6]. This point is nicely illustrated by a lattice quantum chromodynamics benchmark based on an implementation from the Edinburgh Parallel Computing Centre (EPCC) [2]. In the EPCC implementation, much CPU time was consumed by multiplication of 2-spinors by $SU(3)$ gauge elements; this is equivalent to multiplying 3x2 and 3x3 complex matrices. Using the Blitz++ library, one can use the class `TinyMatrix<T,N,M>` for the spinors and gauge elements. This class uses template metaprograms to create specialized algorithms for matrix multiplication. The initial version of the benchmark was this:

```
typedef TinyMatrix<complex<double>,3,3>
    gaugeElement;
typedef TinyMatrix<complex<double>,3,2>
    spinor;

void qcdCalc(Vector<spinor>& res,
    Vector<gaugeElement>& M,
    Vector<spinor>& src)
{
    for (int i=0; i < res.length(); ++i)
        res[i] = product(M[i], src[i]);
}
```

To tune this version, the gauge element and related spinors were encapsulated as a single object:

```
class latticeUnit {
...
protected:
    spinor one;
    gaugeElement ge;
    spinor two;
};
```

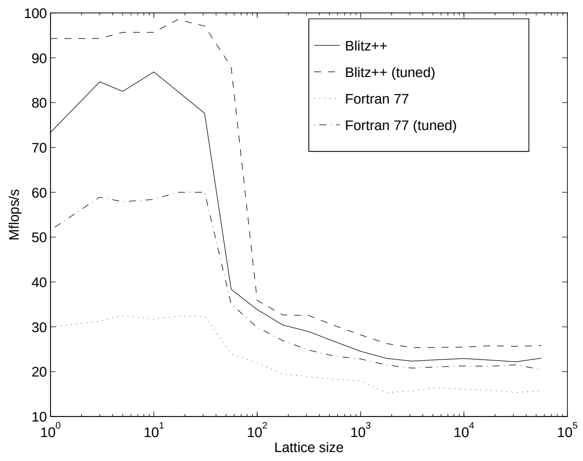

Fig. 2. Results for the lattice QCD benchmark

Figure 2 shows the resulting performance for the initial and tuned Blitz++ versions, as well as a Fortran 77 version. Encapsulation improved the performance by about 12% in the out-of-cache region. Despite extensive hand-tuning of the Fortran version (experimenting with loop-unrolling combinations), it was unable to achieve the performance of either C++ versions due to register spillage.

4 Concluding remarks

C++ is now ready to compete with the performance of Fortran. Its performance problems have been solved by a combination of better optimizing compilers (such as KAI C++) and template techniques.

It's possible that C++ will be faster than Fortran for some applications, since:

- Fortran 90 has the array abstraction built into the language. This limits the ability of the compiler to generate efficient code to expressions involving simple (e.g. dense, asymmetric) arrays. The expression templates technique

in C++ has no such limitations; it can be applied to more complicated (e.g. sparse, jagged-edged) arrays.
- Template metaprograms can generate specialized algorithms well beyond the ability of current Fortran compilers.
- The generic programming features of C++ allow the creation of generic frameworks for common scientific computing tasks (such as solving finite-differenced PDEs). Such frameworks can implement complex optimizations (such as iteration-space tilings).
- The object-oriented nature of C++ encourages favourable memory access patterns.

5 Acknowledgments

This work was supported in part by the National Science and Engineering Research Council of Canada, and by Rogue Wave Software of Corvallis, Oregon. Axel Thimm suggested useful improvements to the QCD example. Arch Robison and colleagues at KAI provided helpful advice on the use of their compiler.

References

1. Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *Computer*, 23(12):25–37, Dec 1990.
2. Stephen Booth. Lattice QCD simulation programs on the Cray T3D. Technical Report EPCC-TR96-03, Edinburgh Parallel Computing Centre, 1996.
3. Scott Haney. Is C++ fast enough for scientific computing? *Computers In Physics*, 8(6):690–694, Nov/Dec 1994.
4. Rebecca Parsons and Daniel Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference (OON-SKI'94)*, pages 408–418, April 24–27, 1994.
5. Arch D. Robison. The abstraction penalty for small objects in C++. In *POOMA'96: The Parallel Object-Oriented Methods and Applications Conference*, Feb. 28 - Mar. 1 1996. Santa Fe, New Mexico.
6. Arch D. Robison. C++ gets faster for scientific computing. *Computers in Physics*, 10(5):458–462, Sep/Oct 1996.
7. Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
8. Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.
9. Todd Veldhuizen and Kumaraswamy Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal of Software Tools*, 21(8):38–44, August 1996.
10. Michael Wolfe. Iteration space tiling for memory hierarchies. In Gary Rodrigue, editor, *Proceedings of the 3rd Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, December 1989. SIAM Publishers.