TOWARD FORTRAN 77 PERFORMANCE FROM OBJECT-ORIENTED C++ SCIENTIFIC FRAMEWORKS

Federico Bassetti, Kei Davis, and Dan Quinlan
Scientific Computing Group CIC-19
Computing, Information, and Communications Division
Los Alamos NM, USA, 87545
{fede,kei,dquinlan}@lanl.gov

Keywords: performance analysis, program generators.

ABSTRACT

The use of object-oriented C++ frameworks has significantly simplified the development of numerous complex parallel scientific applications at Los Alamos National Laboratory and elsewhere. In spite of considerable use of, and commitment to, these frameworks, concerns about performance are nonetheless a significant issue; performance very close to that of FORTRAN 77 with message passing must be realized before the acceptance and use of such frameworks will be truly widespread. This paper identifies the primary source of inefficiency in using C or C++ for numerical codes with stencil- or stencil-like operations, and demonstrates two solutions—one portable, one not—to give genuine FORTRAN 77 performance.

INTRODUCTION

The use of object-oriented C++ frameworks has significantly simplified the development of numerous complex parallel scientific applications at Los Alamos National Laboratory (LANL) and elsewhere; examples at LANL include OVERTURE (Brown et al. 1997) and POOMA (Reynders, J.V.W. et al. 1996) In spite of considerable use of, and commitment to, these frameworks, concerns about performance are nonetheless a significant issue; simply put, performance very close to that of carefully hand-crafted C or FORTRAN 77 with message passing must be realized before the acceptance and use of such frameworks will be truly widespread—FORTRAN 77 is our baseline for evaluating performance. We posit that comparisons with FORTRAN 90, naive C, or similar are largely meaningless and often misleading. Throughout this paper, FORTRAN means FORTRAN 77.

For numerical applications the array types are the most basic; array-class libraries (such as A++/P++ (Quinlan and Parsons 1994) and POOMA) are the basis of higher-level libraries (such as OVERTURE) so their performance is of paramount importance. OVERTURE, for example, comprises a large object-oriented hierarchy of libraries on top of the array classes to support complex geometry, moving grid computations, visualization, adaptive mesh refinement, grid generation, and more. In this context we are concerned with the performance of both the usual binary operations on arrays (such as addition) and *stencil-like* operations; the latter is the focus of this paper.

The traditional method of implementing array operations in C++ is to overload the built-in binary operators so that, for example, the expression A+B+C, given that A, B and C denote array objects, denotes element-wise addition of the arrays. This entails (at least) two array-class-member function calls, each of which entails a 'for' loop and the creation of a temporary array. A newer technique using the template mechanism of C++ to define so-called *expression templates (ETs)* seeks to eliminate the overhead of overloaded binary operators by using the class template mechanism of C++ to perform transformation on the source code. In essence using ETs allows the function calls to be in-lined and only a single 'for' loop generated. While the ET technique can deliver near-C performance in certain cases, in others it is quite disappointing. A yet-newer technique, which we are actively developing, is to use a source-to-source program-transforming pre-processor. The motivation for the approach is several-fold; what is important here is that this approach gives the finest degree of control and is not limited by the language (here the C++ template mechanism) itself, and does not require that the target language be C++. An earlier paper (Davis, Bassetti, and Quinlan 1997) compares and contrasts these techniques in the context of performance of overloaded binary operators.

This paper focuses specifically on the issue of *how to get genuine FORTRAN performance from stencil-like operations in C++* (ultimately using a pre-processor), and why this performance is not obtained (and perhaps not obtainable) using ET implementations.

METHODOLOGY

The numerical algorithms used for solving partial differential equations (PDEs) are rich with stencil-based computational kernels. In most cases these impose the dominant computational cost of a numerical application. In the solution of PDEs both second- and fourth-order methods are commonly used with stencil widths of three and five, respectively; yet higher order methods are also used, with correspondingly greater stencil widths. In numerical applications, and especially those with complex geometry, the distinguishing characteristic of stencil-like operations is the evaluation of expressions in which the operands are all of the elements within a given 'radius' (the stencil width) of a given element of a *single* array. For example, a three-dimensional array and radius of two gives rise to 125 ($(1+2+2)^3$) operands; higher-order methods give rise to a correspondingly greater number of operands. The term 'stencil-like' is used because the expressions actually evaluated are somewhat simpler than 'real' stencil operations in order to distill the essential run-time behaviour. The expression has the form

```
A[i] =            B[i] +
       W[i+1] * B[i+1] +
       W[i+2] * B[i+2];
```

where `W` is an array of weights, where the nominal number of operands is here is three. The arrays `A`, `B`, and `W` have length 1000. The expression is enclosed in two loops, the inner iterating over `i`, and the outer serving to repeat the inner ten times.

The performance associated with the evaluation of such an expression has been determined to be directly related to the number of register loads entailed. Nonetheless, both the number of loads and the number of machine cycles is given in the benchmarks; the latter because similar loads have different costs because of cache and cache-line effects, and because of instruction-level parallelism (e.g. loads and floating-point addition). Performance data is given in graphical form up to 80 operands. Exact numbers are given in tabular form for the case of three operands, and we show how to accurately predict the number of loads in the general case.

Rather than benchmark an actual ET implementation a conservative emulation is used instead. The emulation has been thoroughly demonstrated to be conservative (Davis, Bassetti, and Quinlan 1997) in the sense that it executes strictly fewer instructions of any given type (and uses strictly fewer machine cycles) than an actual ET implementation. This approach simplifies benchmarking and establishes an upper bound on ET performance.

COMPARISON OF C++, C++ WITH ETS, AND FORTRAN

Figure 1 gives the number of register loads of C++ with (emulated) ETs, straight C++ (essentially ordinary C code), and FORTRAN as a function of the number of operands. The corresponding runtime—number of machine cycles—is shown in Figure 2. The numbers in the particular case of three operands is given following.

The FORTRAN code takes 2/3 the time (machine cycles) of the two C++ codes. Analysis of the measured loads provides an explanation. With FORTRAN the pointers to arrays `A`, `B`, and `W` are loaded once per execution of the outer loop, and the values `w[i+2]`, `B[i+1]`, and `B[i+2]` are reused (as `w[i+1]`, `B[i]`, and `B[i+1]`, respectively) in the subsequent execution of the inner loop body. Thus FORTRAN requires two loads per iteration, yielding 20,000 loads total. The C++ codes require five loads for each execution of the inner loop body, giving 50,000 loads. This is a consequence of the fact that the C (and so C++) standard dictates that pointers are not 'safe' and so cannot be reused. The impact of this repeated loading gives the difference in run-time—machine cycles—shown in Figure 2. The difference is not linear because of pipelining effects and concurrent execution of e.g. floating point instructions and loads.

When the number of operands is increased to the point that there are not enough registers for this reuse register spilling occurs, giving rise to more loads and stores; a more detailed study on register spillage and its impact on optimizations is discussed elsewhere (Davis, Bassetti, and Quinlan 1998).

USING RESTRICT

The repeated reloading of pointers in C (and so C++) is of sufficient general impact on performance that numerous C and C++ compilers provide a keyword `restrict` that in effect instructs the compiler to re-use pointers where possible. The effect of using `restrict` is shown in Figures 3 and 4. Here the C++ code achieves the same performance as FORTRAN, but the performance of C++ with ETs is unchanged. This reflects a general problem with the use of ETs: the intermediate C code is of such complexity that standard compiler optimizations become inapplicable. Data for the 3-operand case is given following.

Implementation	Cycles	Loads
C++	48782	21238
C++/ETs	70673	51201
FORTRAN 77	48970	21243

Besides the fact that it doesn't work with ETs, there are several other problems with the use of `restrict`: it is not part of the C or C++ language standards; it is not uniformly implemented—that is, it has different semantics with different compilers; and it is not universally implemented. At best it is non-portable; at worst it may be dangerous in the sense that it may instruct or permit the compiler to perform transformations that do not preserve correctness.

A GENERAL TRANSFORMATION MECHANISM

As mentioned, the use of ETs is as a transformational mechanism to reduce the inefficiencies inherent in the use of overloaded binary operators, but are themselves here and elsewhere shown to be problematic. We are developing a more general transformational tool (Quinlan and Davis 1997) that permits, with a greater or lesser amount of work, arbitrary transformations to C++ code. In the context of this paper its use is to transform array statements (the inner loop bodies) into separately compilable FORTRAN units which are compiled and linked to the C++ object code. This approach gives the best of both worlds: the full expressiveness of C++ without the performance penalties of computationally intensive array statements in loops. Figures 5 and 6 show the results; following is the data for the 3-operand case.

Implementation	Cycles	Loads
C++	42377	21211
C++/ETs	42923	21180
FORTRAN 77	48970	21243

CONCLUSION

We have shown that for C or C++ codes using arrays, and particularly those that use arrays for stencil-like operations, the C/C++ language standard effectively precludes FORTRAN 77 performance, our touchstone of 'optimality'. The non-standard, non-portable `restrict` feature of many C++ compilers can in simple cases recover this performance difference, but not, for example, when expression templates are used to implement array-class operations. The only apparent general solution, a form of which we have implemented and are further developing, is to use a source-to-source transformational tool that implements these array operations in a language that is as universally implemented as C or C++.

REFERENCES

Brown, D., et al. 1997. *OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments.* Proceedings of the SIAM Parallel Conference (Minneapolis, MN, March 1997).

Reynders, J.V.W. et al. 1996. *POOMA: A Framework for Scientific Simulations on Parallel Architectures.* In Parallel Programming using C++. G.V. Wilson and P. Lu, eds. MIT Press.

Quinlan and Parsons. 1994. *A++/P++ Array Classes for Architecture Independent Finite Difference Computations.* Proceedings of the Second Annual Object-Oriented Numerics Conference (Sunriver, OR, April 1994).

Davis, K.; F. Bassetti, F.; and D. Quinlan. 1997. *A Comparison of Performance-Enhancing Strategies for Parallel Numerical Object-Oriented Frameworks.* Proceedings of the 1997 International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE97) (Marina del Rey, CA, Dec. 8-11).

Davis, K.; F. Bassetti, F.; and D. Quinlan. 1998. *C++ Expression Templates Performance Issues in Scientific Computing.* Proceedings of the 9th Symposium on Parallel and Distributed Processing (SPDP'98, Orlando, FL, March 30-April 3, 1998, (to appear)).

Quinlan, D.; K. Davis. 1997. *ROSE: An Optimizing Preprocessor for the Object-Oriented OVERTURE Framework.* http://www.c3.lanl.gov/ dquinlan/ROSE.html.

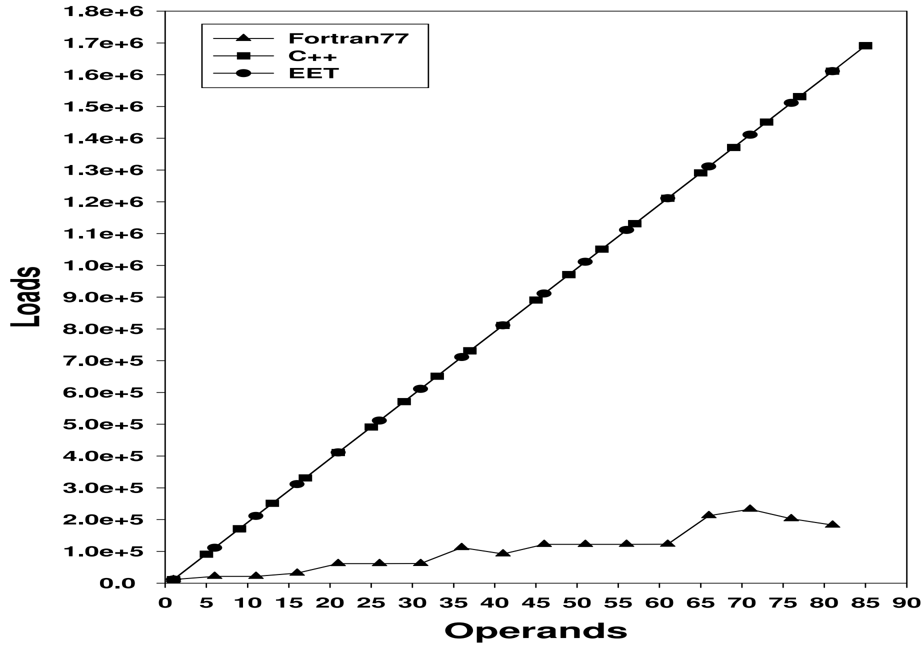

Figure 0: The difference between FORTRAN and C measured in number of loads. FORTRAN has a relatively constant number of loads while for the C it increases with the number of operands. The difference is a result of how each handles pointers. In FORTRAN a pointer is loaded a register and reused when possible. In C a pointer *must* be loaded every time is needed.

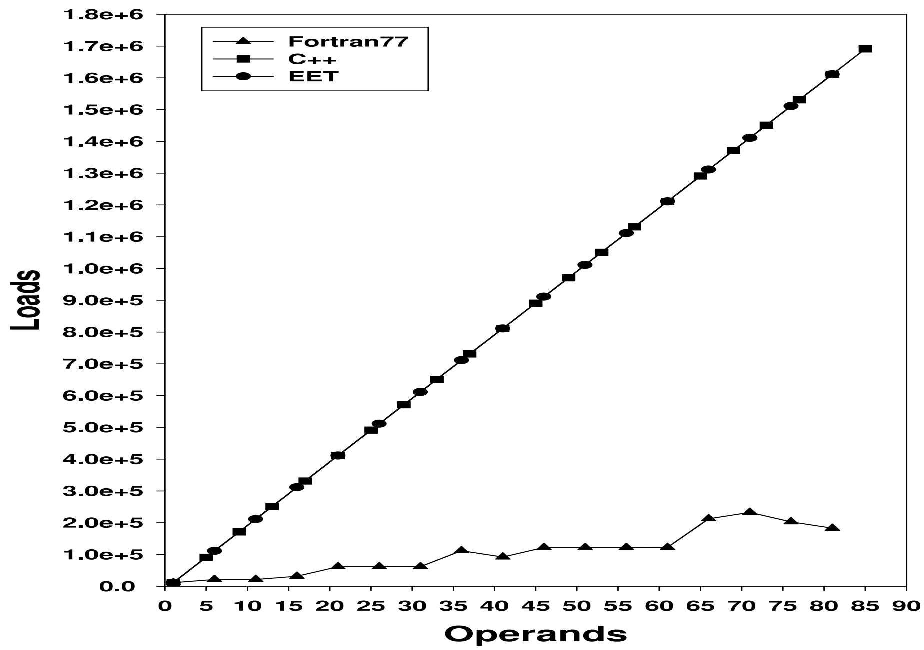

Figure 1: The difference between FORTRAN and C measured in number of loads. FORTRAN has a relatively constant number of loads while for the C it increases with the number of operands. The difference is a result of how each handles pointers. In FORTRAN a pointer is loaded a register and reused when possible. In C a pointer *must* be loaded every time is needed.

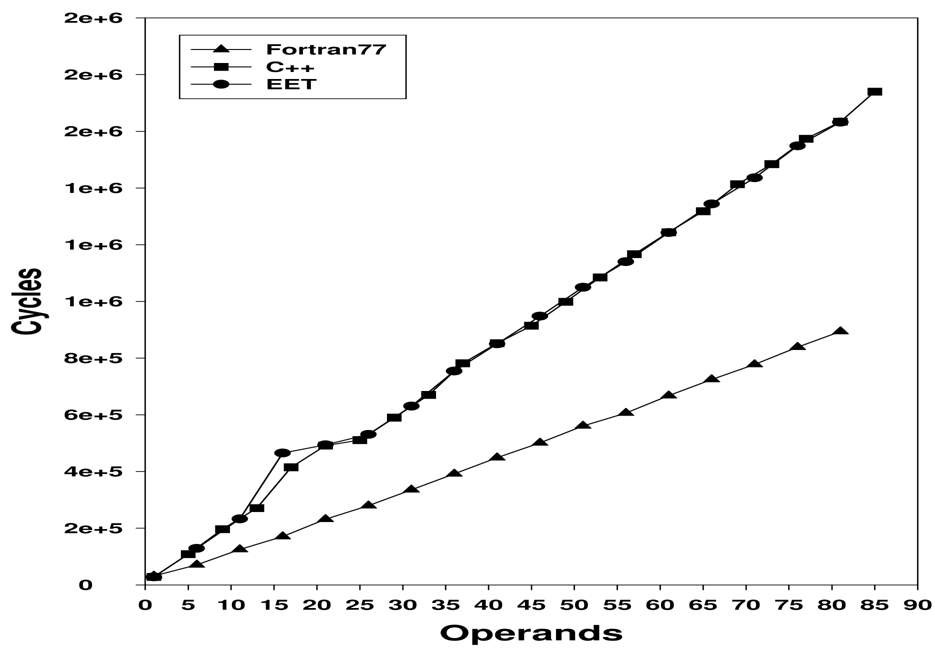

Figure 2: The difference between FORTRAN and C measured in number of cycles. FORTRAN requires fewer cycles than C to perform the same task, almost entirely because FORTRAN requires fewer loads. The effect in terms of cycles is not linear because of pipelining effects and concurrent execution of loads with floating-point operations.

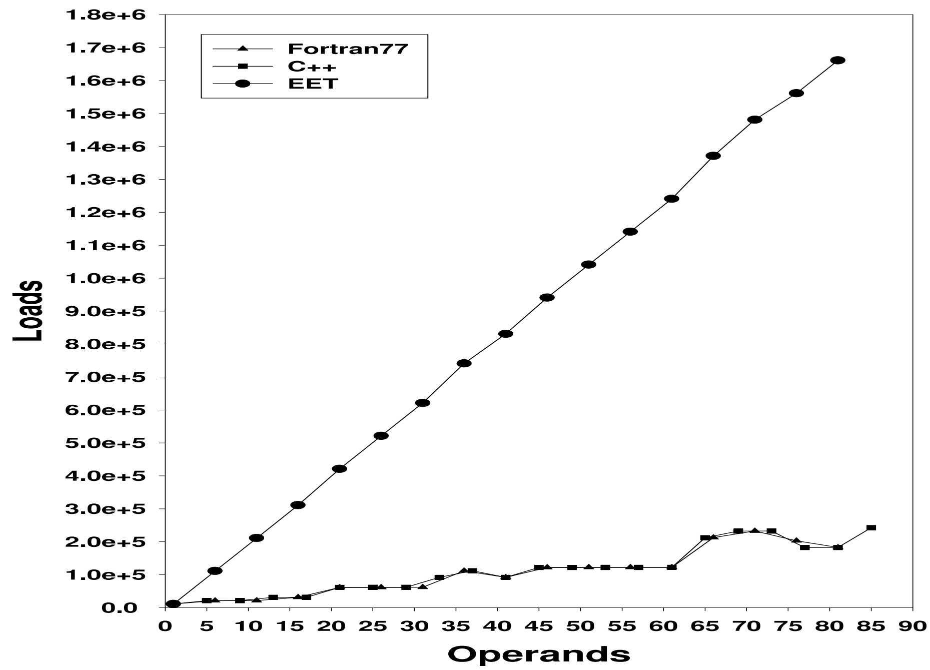

Figure 3: The difference between FORTRAN and C++ with `restrict` measured in loads. C++ with `restrict` is now performs the same number of loads as the FORTRAN code—the pointers are now reused. C++ with ETs does not benefit from the use of `restrict`.

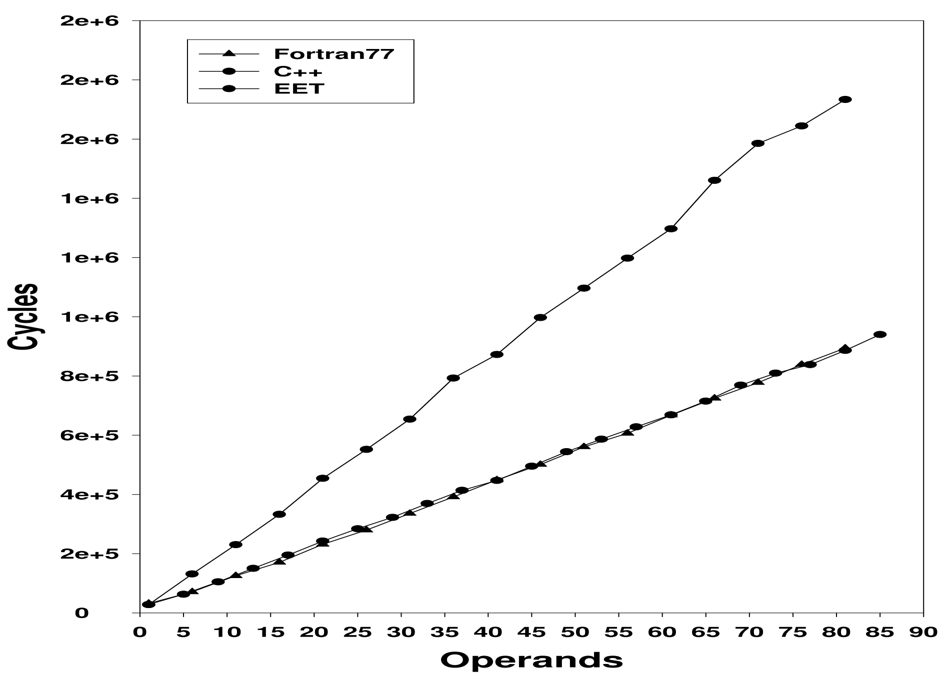

Figure 4: The difference between FORTRAN and C++ enhanced with `restrict` measured in cycles. C++ with `restrict` and FORTRAN have similar execution time.

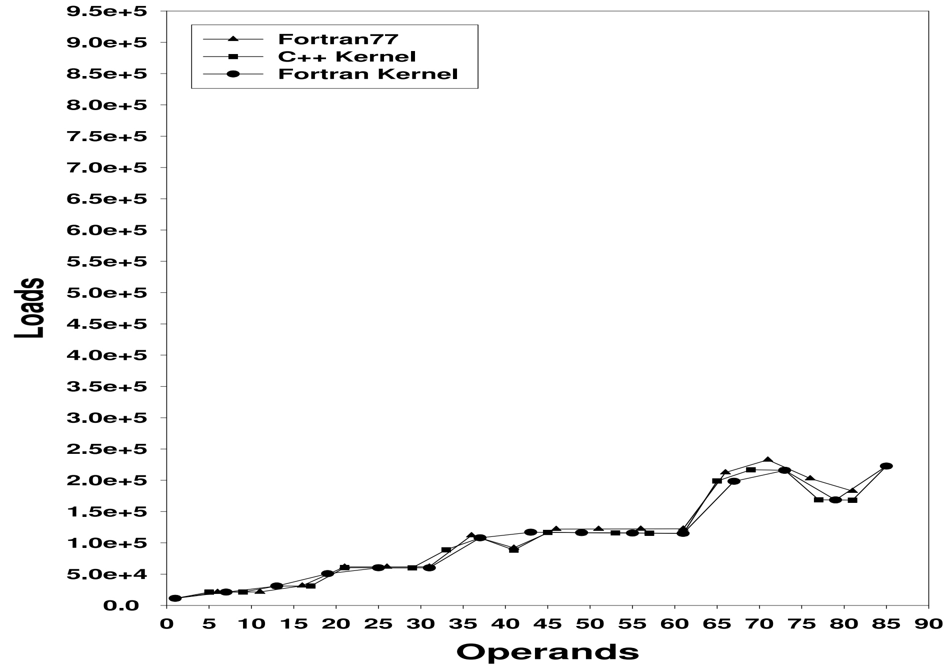

Figure 5: The difference between FORTRAN and C++ calling a FORTRAN kernel. The number of loads shows that the kernel approach leads to the same behavior as FORTRAN. The difference between all the codes in negligible.

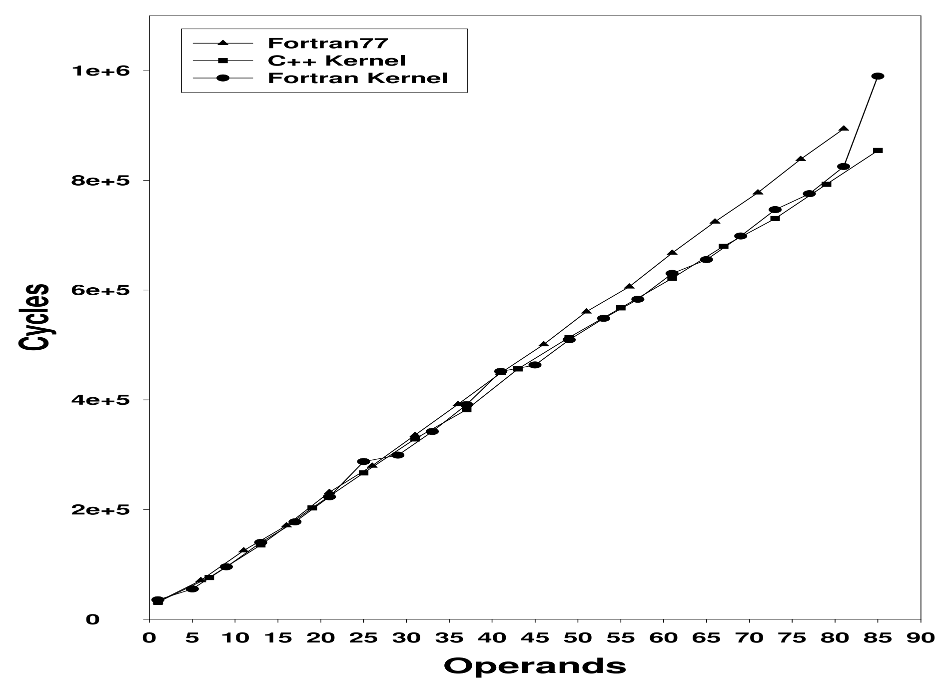

Figure 6: The difference between FORTRAN and C++ calling a FORTRAN kernel. The number of cycles shows that the difference in execution time between the codes are not significant. The kernel approach achieves the same performance as pure FORTRAN.