FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Fast Expression Templates for the Hitachi SR8000 Supercomputer

Alexander Linke and Christoph Pflaum

Technical Report 03/09

Fast Expression Templates for the Hitachi SR8000 Supercomputer

Alexander Linke and Christoph Pflaum

Technical Report 03/09

Fast Expression Templates for the Hitachi SR8000 Supercomputer

Alexander Linke Christoph Pflaum

August 2003

Abstract

Expression templates(ET) can significantly reduce the implementation effort for mathematical software. On the Hitachi SR8000 supercomputer it can be observed, however, that classical ET implementations do not lead to an optimal performance in the case of certain expressions such as $c = aabbab$. This is because they do not assist the compiler in recognizing that variables are used several times within an expression. Therefore, we introduce the concept of enumerated variables, which are provided with an additional integer template parameter. Since different variables have different types, a modified implementation of ET leads to an optimal C++ code on the Hitachi SR8000. These so-called *Fast Expression Templates* perform better than classical ET, even when variables are not used several times. Performance results are presented on the Hitachi SR8000 supercomputer with automatic vectorization and parallelization.

1 Introduction

While object-oriented programming is being embraced in the industry, its acceptance by the High Performing Computing community is still very hesitant, mainly because of supposed performance losses. For example, supercomputer manufacturer Hitachi tells us that there is not a single user of C++ on Hitachi supercomputers in Japan. Indeed, introducing abstract data types and operator overloading empowers the software engineer to forge entire user-defined languages based on C++, which can be understood by C++ compilers. However, there are not any language features in C++ which are designed to inform the compiler about allowed transformations of mathematical expressions involving user-defined abstract data types. Therefore, though such user-defined mathematical expressions can be compiled by C++-compilers, they perform very poorly. The first solution to overcome these performance problems were expression templates in C++, whose development we will summarize in the following.

By adding templates to C++, the language gained much more than originally intended. Todd Veldhuizen recognized the potential of this facility and in 1995 and 1996 published the first articles about template meta programming [11] and expression templates [12]. Expression templates were proposed to overcome performance problems which arise from simple operator overloading in mathematical expressions. Unnecessary temporaries are avoided by performing some kind of expression dependent inlining within a single loop. For many systems the performance of expression templates competes with Fortran [13].

Soon, there was a rapid development of powerful mathematical packages based on expression templates, e.g. Blitz++ by Todd Veldhuizen [14] and the Generative Matrix Computation Library(GMCL) described in [4]. Further important steps were PETE [7] and POOMA [10], which are two projects started at the Los Alamos National Laboratories. PETE is a tool for implementing expression templates for various applications. POOMA supports the implementation of mathematical algorithms for solving partial differential equations. Users of POOMA write sequential source code similar to FORTRAN 90 and get automatic parallelization on various platforms just by using compiler switches.

Some aims of the POOMA project are shared by the EXPDE project [8]. EXPDE eases the implementation of parallel 3D finite elements codes by providing finite element and multigrid operators on arbitrarily shaped domains. Mathematical algorithms can be formulated with EXPDE

in a language very close to the mathematical language. Expression templates enable automatic parallelization.

Performance problems with expression templates were discovered for the first time by Federico Bassetti, Kei Davis and Dan Quinlan in 1997, see [2] and [3].

In this article we want to present another problem with expression templates. The solution of this new problem solves prior problems, too. Let us look at a bit of C++ code with $a*b$ understood as component-wise multiplication:

```
Vector a, b, c;
c = a*a*b*b*a*b;
```

On a single node of the Hitachi supercomputer SR8000 a classical expression templates implementation of this code - as suggested in [15] - achieves a performance significantly inferior to handcrafted C code, although the expression templates code is fully vectorized and parallelized, see figure 4.1.

Generally, expression templates derive a so-called parse tree or expression object representing an expression and perform some expression dependent inlining. Classical implementations split the information about an expression into two parts. The data type of the expression object represents the structure of the expression and member variables represent the variables involved within the expression. The data type of the expression object is known at compile-time. However, use of information represented by member variables is compiler-dependent. For example, information accessible at compile-time about the expression `a*a*b*b*a*b` is equivalent to:

```
Vector*Vector*Vector*Vector*Vector*Vector
```

Therefore, classical expression templates do not assist the compiler in recognizing that the Vectors a and b are used several times within the expression and performance losses can arise.

This problem can be overcome by introducing the concept of enumerated variables. Enumerated variables are provided with an additional integer template parameter. Since different variables have different types, the compiler can perform more intelligent expression dependent inlining. Moreover, by using enumerated variables, the intermediate C++ code can be forced to be identical to handcrafted C code and performance measurements on the Hitachi SR8000 show optimal C++ performance. Even all the performance issues recognized by Bassetti, Davis and Quinlan do not occur with our new implementation of expression templates.

1.1 Structure of the Paper

In the next section we will explain in detail how expression templates are classically implemented, how they work internally and which problems they suffer from. Thereafter, we present in detail a suggestion for a modified implementation. In the last two sections we show performance results and draw conclusions about the use of expression templates in High Performance Computing.

2 Classical Expression Templates

Let us discuss a minimal classical expression templates implementation for vectors with component-wise multiplication, as suggested in [15]. Our example was implemented on the Hitachi SR8000 supercomputer. The program runs on a single node of the Hitachi, which is equipped with 8 processors. Each processor has so-called pseudo-vectorization facilities. For our presentation it is enough to imagine that each processor is a vector-processor. Parallelization and vectorization is enabled by the C99 keyword `restrict` and the compiler directive `/*voption indep*/`, see [5] and [6]. The mathematical operator `times` is encapsulated by:

```
struct times {
public:
  static inline double apply(double a, double b) {
    return a*b;
  }
};
```

The expression class is implemented as:

```
template<typename Left, typename Op, typename Right>
struct Expr {
  const Left &leftNode_;
  const Right &rightNode_;

  Expr(const Left &t1, const Right &t2)
    : leftNode_(t1), rightNode_(t2) {}

  inline double Give(int i) const {
    return Op::apply(leftNode_.Give(i), rightNode_.Give(i));
  }
};
```

Here is a simple vector class:

```
struct Vector {
  Vector(double *restrict &data, int N) : data_(data), N_(N) {}

  template<typename Left, typename Op, typename Right>
  inline void operator=(const Expr<Left, Op, Right> &expression) {
    int N = N_;

    /*voption indep*/
    for(int i=0; i < N; ++i)
      data_[i] = expression.Give(i);
  }

  inline double Give(int i) const {
    return data_[i];
  }

  double * &data_;

  int N_;
};
```

and here the `operator*`:

```
template<typename Left>
Expr<Left, times, Vector> operator*(const Left &a,
  const Vector &b) {
  return Expr<Left, times, Vector >(a, b);
}
```

Now wee see it in action:

```
  ...
  Vector
    a(a_data, N), b(b_data, N), r(r_data, N);
  r = a*b*a;
  ...
```

We will now explain how expression templates work. Let us look at the line `r = a*b*a;`. The simulated run of the compiler looks like:

```
  r = a*b*a;
    = Expr<Vector,times,Vector>(a,b)*a;
    = Expr<Expr<Vector,times,Vector>,times,Vector>(
        Expr<Vector,times,Vector>(a,b),a);
    =: expression;
```

It then matches `r.operator=`:

```
r.operator=<Expr<Expr<Vector,times,Vector>,times,Vector> >
  (expression) {
  int N = N_;

  /*voption indep*/
  for(int i=0; i < N; ++i)
    data_[i] = expression.Give(i);
}
```

Now `expression.Give(i)` is expanded by inlining `Give(i)` from each node of the expression object:

```
data_[i] = expression.Give(i);
         = times::apply(expression.leftNode_.Give(i),
                        expression.rightNode_.Give(i));
         = times:apply(times::apply(
             expression.leftNode_.leftNode_.Give(i),
               expression.leftNode_.rightNode_.Give(i)),
             expression.rightNode_.Give(i));
         = times::apply(times::apply(
             expression.leftNode_.leftNode_.data_[i],
               expression.leftNode_.rightNode_.data_[i]),
             expression.rightNode_.data_[i]);
         = expression.leftNode_.leftNode_.data_[i] *
             expression.leftNode_.rightNode_.data_[i] *
           expression.rightNode_.data_[i];
```

Although component-wise multiplication of three vectors is a very trivial application of expression templates, the intermediate C++ code inlined by expression templates is quite complex. More difficult applications like expression templates for differential operators in 3D yield even more complex intermediate code. If `operator=` is inlined, a C++ compiler can, in principle, optimize the above expression in the sense that it evolves to:

```
for(int i=0; i < N; ++i)
  r.data_[i] = a.data_[i] * b.data_[i] * a.data_[i];
```

Indeed, this requires a compiler to have good optimization facilities. Since the Hitachi C++ compiler does not cope with C++-specific optimization facilities, on the Hitachi SR8000 classical expression templates suffer from several performance problems.

3 Fast Expression Templates

After this short discussion of classical expression templates, we present an alternative approach. The main idea is to derive an expression object from an expression whose information is completely accessible at compile-time. This allows more intelligent inlining and guarantees optimal C++ performance. We achieve this by introducing enumerated variables. Enumerated variables have an additional integer template parameter, which has to be chosen uniquely for each variable. Then different variables have different types. As before, we present a minimal implementation of this new approach. This causes the examples to be somewhat cumbersome, but we feel that it makes understanding the approach easier.

The presented implementation restricts us to use only three different enumerated variables, but is easily extended to an arbitrarily high number of enumerated variables. An alternative, more elegant, approach would use typelists, as described in [1]. In this case there are no restrictions on the number of enumerated variables. Global variables are easily avoided by additionally introducing expression wrapper classes, which wrap a single enumerated variable within an expression.

First we introduce some macros, which simplify the handling of enumerated variables:

```
#define params_in double *restrict data0_, \
                  double *restrict data1_, \
                  double *restrict data2_
#define params_out data0_, data1_, data2_
#define DeclareEnumVariables double *restrict data0_, \
                                    *restrict data1_, \
                                    *restrict data2_

double *global0_, *global1_, *global2_;
```

The classes, which derive the expression object from a mathematical expression, have the same structure as in the classical case. However, they do not have any member variables and the methods can be defined as `static`. The complete information about an expression is now represented by its data type.

```
struct times {
public:
  static inline double apply(double a, double b) {
    return a*b;
  }
};

template<typename Left, typename Op, typename Right>
struct Expr {
  static inline double Give(params_in, int i) {
    return Op::apply(Left::Give(params_out, i),
                     Right::Give(params_out, i));
  }
};
```

The main implementation idea consists of introducing the enumerated variable class `vector`. The expression dependent inlining is again performed by methods defined as `static`. Furthermore, it is remarkable that the new approach allows to abandon the compiler directive `/*voption indep*/`. We achieve parallelization and vectorization just by using compiler switches and the C99 keyword `restrict`:

```
template<int varnum>
struct Vector {
  template<typename Left, typename Op, typename Right>
  void operator=(const Expr<Left, Op, Right> &expression) const {
    DeclareEnumVariables;
    // double *restrict data0_, *restrict data1_, ...;

    int N = global_N;
    // Preparations, can be implemented in a more elegant manner
    data0_ = global0_; data1_ = global1_; data2_ = global2_;

    for(int i=0; i < N; ++i)
      GiveData(params_out)[i] = expression.Give(params_out, i);
  }

  static inline double Give(params_in, int i) {
    return 0.0;
  }

  static inline double*restrict GiveData(params_in) {
    return NULL;
  }
```

```
};

template<> inline double Vector<0>::Give(params_in, int i) {
  return data0_[i]; }
template<> inline double Vector<1>::Give(params_in, int i) {
  return data1_[i]; }
template<> inline double Vector<2>::Give(params_in, int i) {
  return data2_[i]; }

template<>
inline double*restrict Vector<0>::GiveData(params_in) {
  return data0_; }
template<>
inline double*restrict Vector<1>::GiveData(params_in) {
  return data1_; }
template<>
inline double*restrict Vector<2>::GiveData(params_in) {
  return data2_; }
```

The following piece of code is analogous to the classical code:

```
template<typename Left, typename Right>
inline Expr<Left, times, Right> operator*(const Left &a,
  const Right &b) {
  return Expr<Left, times, Right>();
}

template<typename Left, int varnum>
Expr<Left, times, Vector<varnum> > operator*(const Left &a,
  const Vector<varnum> &b) {
  return Expr<Left, times, Vector<varnum> >(a, b);
}
```

Last but not least, an excerpt from the main program:

```
  ...
  Vector<0> a; Vector<1> b; Vector<2> r;
  r = a*b*a;
  ...
```

We will now explain how this new approach differs from classical expression templates. To this end, let us look at the expression `c=a*b*a`. The expression object for this expression has the following type:

```
Expr<Expr<Vector<0>,times,Vector<1> >,times,Vector<0> >
```

Since this expression type contains the complete information about the expression `a*b*a`, a more intelligent inlining is possible. The expression is evaluated as:

```
  r.operator=Expr<Expr<Vector<0>,times,Vector<1> >,times,
              Vector<0> >(expression) {
    DeclareEnumVariables;
    int N = global_N;
    // Preparations, can be implemented in a more elegant manner
    data0_ = global0_; data1_ = global1_; data2_ = global2_;

    for(int i=0; i < N; ++i)
      GiveData(params_out)[i] = expression.Give(params_out, i);
  }
```

Now we investigate how `expression.Give(params_out, i)` is inlined:

```
data2_[i] = times:apply(
          times::apply(
            expression::Left::Left::Give(params_out, i),
            expression::Left::Right::Give(params_out, i)),
          expression::Right::Give(params_out, i));
     = times:apply(
          times::apply(data0_[i], data1_[i]), data0_[i]);
     = data0_[i]*data1_[i]*data0_[i];
```

This inlined source code is just the code a programmer would naively write. Since `expression::Left::Left` and `expression::Right` are of type `Vector<0>`, and since `expression::Left::Right` is of type `Vector<1>`, enumerated variables allow the most precise static inlining. This static inlining is very important, because it assures that the expression object is created at compile-time, not at run-time. This has an important impact on performance.

The new approach for expression templates presented here is much more powerful than classical implementations. This is not only true for our trivial minimal implementation with component-wise multiplication, but also for complex expression templates applications on 3D-grids with 3D-stencil-evaluations.

4 Performance Results

In this section we will present performance results for three different vector expressions on a single node of the Hitachi SR8000 supercomputer in Munich. A single node is equipped with eight RISC processors and each processor can perform vector operations with floating point numbers. The peak performance of a Hitachi node is 12 GFLOPS per second, see [5].

On the Hitachi SR8000 we use the Optimizing C++ compiler sCC by Hitachi, because it is the only C++ compiler which can vectorize C++ programs on this platform. Unfortunately, this compiler is available only as a beta version 5. It has some bugs and is not fully ANSI C++ compliant. However, it optimizes numerical computations, actually written in C code, fairly well. Examples were compiled with the compiler options `-restrict -Os`. The keyword `restrict` is part of ANSI C99, see [5].

For each example we compare three different implementations. The first consists of handcrafted C code compiled by a C++ compiler, called *No Expression Templates*(NET). The second is a *Classical Expression Templates*(CET) implementation and the third is our new approach, called *Fast Expression Templates*(FET). For performance measurement of our FET implementation we used a minimal implementation with five variables and component-wise vector addition and multiplication. Performance measurements are presented for the three following examples:

$$c = a*a*b*b*a*b \tag{1}$$

$$a = a+a*a+a*a*a+a*a*a*a+a*a*a*a*a+ a*a*a*a*a*a+a*a*a*a*a*a*a \tag{2}$$

$$a = b*c+d*e \tag{3}$$

Each expression was evaluated 1000 times and the elapsed time was measured. In the case of NET and FET code the sCC compiler can optimize away superfluous floating point operations, but not in case of CET.

4.1 First Example

The first example $c = a*a*b*b*a*b$ makes clear that FET is better in advising the sCC compiler that variables are used several times within an expression. NET and FET code achieve the same performance. The logfiles created by sCC concerning loop unrolling, parallelization and vectorization, show that the intermediate C++ code inlined by FET is compiled very similar to NET code. Performance of CET is measurably lower than of NET or FET.

The following two tables show the performance improvement achieved by FET in comparision to CET for (1) and the absolute performance of FET measured in units of MFLOPS per second.

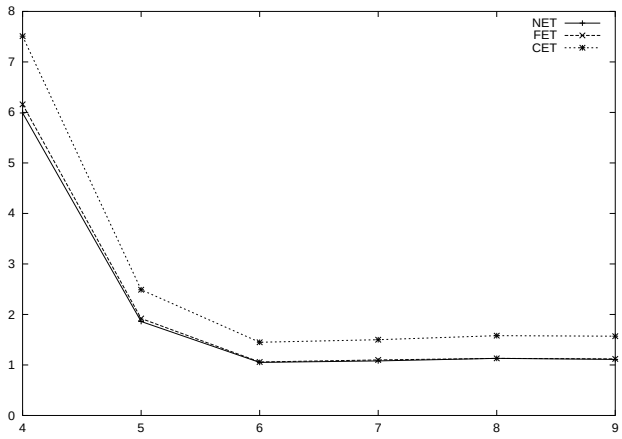

Figure 1: Elapsed time per vector component for three implementations of the expression $c = a*a*b*b*a*b$ on a single node of the Hitachi SR8000. With N the vector length, we plot elapsed time per vector component against $\log_5 N$. The abbreviations NET, CET and FET stand for *No Expression Templates*, *Classical Expression Templates* and *Fast Expression Templates*

$\log_5$(*Vector length*)	4	5	6	7	8	9
Performance ratio FET/CET	1.22	1.30	1.37	1.36	1.40	1.40

$\log_5$(*Vector length*)	4	5	6	7	8	9
Performance of FET	812	2600	4720	4550	4430	4460

4.2 Second Example

The second example

$$a = a + a*a + a*a*a + a*a*a*a + a*a*a*a*a + a*a*a*a*a + a*a*a*a*a*a + a*a*a*a*a*a*a$$

again makes clear that FET is better in advising the sCC compiler that variables are used several times within an expression. However, in this example the sCC compiler can exploit this much more. Hardware counters show that, with N the vector length, NET and FET perform $12N$ floating point operations to evaluate the expression once. According to the Horner scheme this is the minimum number of operations needed. Unlike NET and FET, the CET implemention performs $27N$ floating point operations, since in this case the sCC does not recognize that the variable a is used several times in this expression.

Again, NET and FET achieve the same performance and their logfiles are identical.

The following two tables show the performance improvement achieved by FET in comparision to CET and the absolute performance of FET measured in units of MFLOPS per second.

$\log_5$(*Vector length*)	4	5	6	7	8	9
Performance ratio FET/CET	3.50	5.86	7.46	8.04	9.05	9.14

$\log_5$(*Vector length*)	4	5	6	7	8	9
Performance of FET	1390	3880	5730	6340	6350	6420

This example shows that for the sCC compiler there are expressions, for which FET will perform arbitrarily better than CET.

4.3 Third Example

The third example shows that FET perform better then CET, even when variables are not used several times within an expression. Therefore, FET seem to solve the problems recognized by Bassetti, Davis and Quinlan in [3], too.

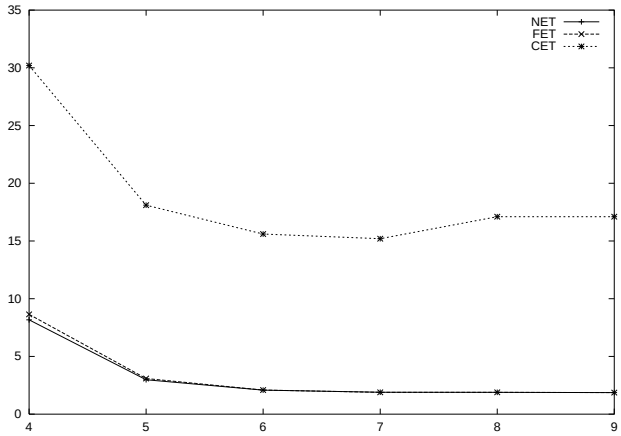

Figure 2: Elapsed time per vector component for three implementations of the expression $a = a + a * a + a * a * a + \cdots + a * a * a * a * a * a * a * a$ on a single node of the Hitachi SR8000. For explanations, see figure 4.1

.

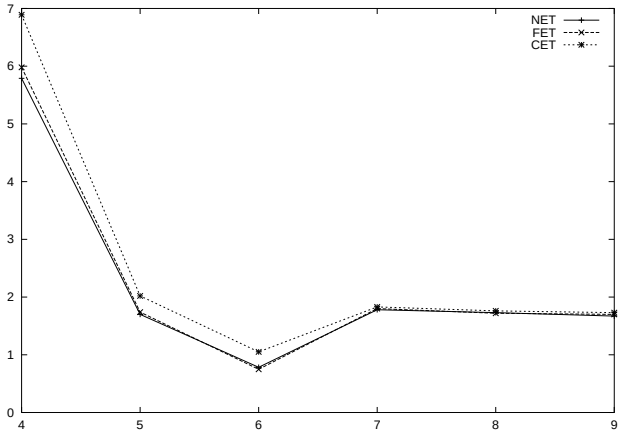

Figure 3: Elapsed time per vector component for three implementations of the expression $a = b * c + d * e$ on a single node of the Hitachi SR8000. For explanations, see figure 4.1

The following two tables show the performance improvement achieved by FET in comparision to CET and the absolute performance of FET measured in units of MFLOPS per second.

$\log_5(\textit{Vector length})$	4	5	6	7	8	9
Performance ratio FET/CET	1.15	1.16	1.40	1.02	1.02	1.02

$\log_5(\textit{Vector length})$	4	5	6	7	8	9
Performance of FET	501	1720	4010	1660	1740	1770

5 Conclusions and Perspectives

In this article we wanted to show that classical implementations of expression templates exploit the potential of the approach only partially, its real potential being far from recognized. Expression templates should be investigated further in many directions. Optimal C++ performance is possible if expression templates are implemented in an optimal way. Since the complete information represented by an expression object is accessible at compile-time, optimal C++ performance should be achievable also for more complex expression templates applications like 3D finite elements expressions. Therefore, the prospects of expression templates seem to be very interesting. Two possible new applications shall be outlined in short.

First, let us look at our 3D finite element codes on the Hitachi SR8000. At the upper level of abstraction it is possible to combine expression templates with template meta programming. We can then define complex transformation rules for expressions by template meta programming in order to achieve high-level optimizations. An expression like `DxDx_FE(u) + DyDy_FE(u) + DzDz_FE(u)` could be transformed by template meta programming into `Laplace_FE(u)`.

At the machine-oriented level it should, with expression templates, be possible to calculate the number of vectorization streams necessary for any particular mathematical expression. If the number of necessary streams exceeds the number of available streams, inlining can be controlled such that the entire expression is divided into several subexpressions. Then, although temporaries are generated, such expressions perform better on the Hitachi SR8000 due to full vectorization.

Such automatic optimization facilities introduce modularity in the implementation of mathematical software. The formulation of algorithms can be close to the usual mathematical language. Optimization, parallelization and vectorization of mathematical expressions can be masked completely by expression templates implementation. This seems to us a great gain.

References

[1] Alexandrescu, A: Modern C++ Design : Generic Programming and Design Patterns applied. Addison-Wesley, Boston, 2001.

[2] Bassetti, F, Davis, K, and Quinlan, D: Toward Fortran 77 Performance from Object-Oriented C++ Scientific Framework: HPC '98 April 5-9, 1998.

[3] Bassetti, F, Davis, K, and Quinlan, D: C++ Expression Templates Performance Issues in Scientific Computing. CRPC-TR97705-S, October 1997.

[4] Czarnecki, K, and Eisenecker, U: Generative Programming : Methods, Tools, and Applications. Addison-Wesley, Boston, 2000.

[5] Leibniz-Rechenzentrum München: The Hitachi SR8000-F1, System Description. http://www.lrz-muenchen.de/services/compute/hlrb/system-en

[6] Numerical C Extensions Group: Restricted Pointers in C. Final Report, Draft 2, X3J11/94-019, WG14/N334, http://www.lysator.liu.se/c/restrict.html

[7] Los Alamos National Laboratories: PETE - Portable Expression Templates Engine. http://www.acl.lanl.gov/pete/html/introduction.html

[8] Pflaum, C: Expression Templates for Partial Differential Equations. Comput Visual Sci **4**, 1–8, (2001).

[9] Pflaum, C, and Falgout, R: Automatic Parallelization with Expression Templates. Lawrence Livermore National Laboratory technical report UCRL-JC-146179, November 2001.

[10] Los Alamos National Laboratories: POOMA: www.acl.lanl.gov/pooma

[11] Veldhuizen, T: Using C++ Template Metaprograms. *C++ Report* Vol. 7 No 4. (May 1995), pp. 36–43.

[12] Veldhuizen, T: Expression Templates. C++ Report **7** (5), 26–31 (1995).

[13] Veldhuizen, T: Will C++ be faster than Fortran? Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97).

[14] Veldhuizen, T: Blitz++. http://oonumerics.org/blitz/index.html

[15] Veldhuizen, T: Techniques for Scientific C++. Indiana University Computer Science Technical Report No 542, Version 0.4, August 2000.