Scientific Programming 11 (2003) 273–290
IOS Press

Implementing a high performance tensor library

Walter Landry
University of Utah, University of California San Diego, 9500 Gilman Dr, Mail Code 0424, La Jolla, CA 92093-0424, USA
Tel.: +1 858 822 5053; E-mail: wlandry@ucsd.edu

Abstract. Template methods have opened up a new way of building C++ libraries. These methods allow the libraries to combine the seemingly contradictory qualities of ease of use and uncompromising efficiency. However, libraries that use these methods are notoriously difficult to develop. This article examines the benefits reaped and the difficulties encountered in using these methods to create a friendly, high performance, tensor library. We find that template methods mostly deliver on this promise, though requiring moderate compromises in usability and efficiency.

1. Introduction

Tensors are used in a number of scientific fields, such as geology, mechanical engineering, and astronomy. They can be thought of as generalizations of vectors and matrices. Consider the rather prosaic task of multiplying a vector P by a matrix T, yielding a vector Q

$$\begin{pmatrix} Q_x \\ Q_y \\ Q_z \end{pmatrix} = \begin{pmatrix} T_{xx} & T_{xy} & T_{xz} \\ T_{yx} & T_{yy} & T_{yz} \\ T_{zx} & T_{zy} & T_{zz} \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix}.$$

If we write out the equations explicitly then

$$Q_x = T_{xx}P_x + T_{xy}P_y + T_{xz}P_z,$$
$$Q_y = T_{yx}P_x + T_{yy}P_y + T_{yz}P_z,$$
$$Q_z = T_{zx}P_x + T_{zy}P_y + T_{zz}P_z.$$

Alternatively, we can write it as

$$Q_x = \sum_{j=x,y,z} T_{xj}P_j$$
$$Q_y = \sum_{j=x,y,z} T_{yj}P_j$$
$$Q_z = \sum_{j=x,y,z} T_{zj}P_j$$

or even more simply as

$$Q_i = \sum_{j=x,y,z} T_{ij}P_j,$$

where the index i is understood to stand for x, y, and z in turn. In this example, P_j and Q_i are vectors, but could also be called rank 1 tensors (because they have one index). T_{ij} is a matrix, or a rank 2 tensor. The more indices, the higher the rank. So the Riemann tensor in General Relativity, R_{ijkl}, is a rank 4 tensor, but can also be envisioned as a matrix of matrices. There are more subtleties involved in what defines a tensor, but it is sufficient for our discussion to think of them as generalizations of vectors and matrices.

Einstein introduced the convention that if an index appears in two tensors that multiply each other, then that index is implicitly summed. This mostly removes the need to write the summation symbol $\sum_{j=x,y,z}$. Using this Einstein summation notation, the matrix-vector multiplication becomes simply

$$Q_i = T_{ij}P_j.$$

Of course, now that the notation has become so nice and compact, it becomes easy to write much more complicated formulas such as the definition of the Riemann tensor

$$R^i_{jkl} = dG^i_{jkl} - dG^i_{lkj} + G^m_{jk}G^i_{ml} - G^m_{lk}G^i_{mj}.$$

There are some subtle differences between tensors with indices that are upstairs (like T^i), and tensors with

ISSN 1058-9244/03/$8.00 © 2003 – IOS Press. All rights reserved

indices that are downstairs (like T_i), but for our purposes we can treat them the same. Now consider evaluating this equation on an array with N points, where N is much larger than the cache size of the processor. We could use multidimensional arrays and start writing lots of loops

```
for(int n=0;n<N;++n)
 for(int i=0;i<3;++i)
  for(int j=0;j<3;++j)
   for(int k=0;k<3;++k)
    for(int l=0;l<3;++l)
     {
      R[i][j][k][l][n]=dG[i][j][k][l][n]
                      - dG[i][l][k][j][n];
      for(int m=0;m<3;++m)
        R[i][j][k][l][n]+=G[m][j][k][n]
                         * G[i][m][l][n]
                         - G[m][l][k][n]
                         * G[i][m][j][n];
     }
```

This is a dull, mechanical, error-prone task, exactly the sort of thing we want computers to do for us. This style of programming is often referred to as C-tran, since it is programming in C++ but with all of the limitations of Fortran 77. We would like to write something like

```
R(i,j,k,l)=dG(i,j,k,l) - dG(i,l,k,j)
           + G(m,j,k)*G(i,m,l) - G (m,l,k)
           * G(i,m,j);
```

and have the computer do all of the summing and iterating over the grid automatically.

There are a number of libraries with varying amounts of tensor support [1–3,8–11]. With one exception, they are all either difficult to use (primarily, not providing implicit summation), or they are not efficient. GRPP [8] solves this conundrum with a proprietary mini-language, making it difficult to customize and extend.

We have written a program to simulate neutron star collisions in General Relativity. It uses tensors extensively, so we have developed a library to simplify their use. In this paper, we start by describing our first, simple design for a tensor class within C++. We progressively refine the overall design to improve the performance, while keeping most of the usability intact. Then we describe the details of implementing natural notation for tensor arithmetic within this final design. We follow with a survey of different compilers, testing how proficient they are at compiling and optimizing the library. We end with a look at a more generic version of the library and how it affects performance.

2. Design choices for tensor libraries

There are a few different ways that a tensor library can be constructed. Ideally, we want a solution that is easy to implement, easy to use, and efficient.

2.1. Simple classes

The most straightforward way to proceed is to make a set of classes (Tensor1, Tensor2, Tensor3, etc.) which simply contains arrays of doubles of size N. Then we overload the operators $+$, $-$ and * to perform the proper calculation and return a tensor as a result. The well known problem with this is that it is slow and a memory hog. For example, the expression

$$A_i = B_i + C_i\,(D_j E_j)\,,$$

will generate code equivalent to

```
double *temp1=new double [N];
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    temp1[n]=D[i][n]*E[i][n];
double *temp2[3]
temp2[0]=new double[N];
temp2[1]=new double[N];
temp2[2]=new double[N];
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    temp2[i][n]=C[i][n]*temp1[n];
double *temp3[3]
temp3[0]=new double[N];
temp3[1]=new double[N];
temp3[2]=new double[N];
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    temp3[i][n]=B[i][n]+temp2[i][n];
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    A[i][n]=temp3[i][n];
delete[] temp1;
delete[] temp2[0];
delete[] temp2[1];
delete[] temp2[2];
delete[] temp3[0];
delete[] temp3[1];
delete[] temp3[2];
```

This required three temporaries $(\mathrm{temp1} = D_j E_j)$, $(\mathrm{temp2}_i = C_i * \mathrm{temp1})$, $(\mathrm{temp3}_i = B_i + \mathrm{temp2}_i))$ requiring 7 N doubles of storage. None of these temporaries disappear until the whole expression finishes. For expressions with higher rank tensors, even more temporary space is needed. Moreover, these temporaries are too large to fit entirely into the cache, where they can be quickly accessed. The temporaries have to be moved to main memory as they are computed, even though they will be needed for the next calculation. With current architectures, the time required to move all of this data back and forth between main memory and the processor is much longer than the time required to do all of the computations.

2.2. Expression templates

This is the sort of problem for which template methods are well-suited. Using expression templates [13], we can write

```
A(i)=B(i)+C(i)*(D(j)*E(j));
```

and have the compiler transform it into something like

```
for(int n=0;n<N;++n)
  for(int i=0;i<3;++i)
    {
      A[i][n]=B[i][n];
      for(int j=0;j<3;++j)
        A[i][n]+=C[i][n]*(D[j][n]*E[j][n]);
    }
```

The important difference here is that there is only a single loop over the N points. The large temporaries are no longer required, and the intermediate results (like `D[j][n]*E[j][n]`) can stay in the cache. This is a specific instance of a more general code optimization technique called loop-fusion. It keeps variables that are needed for multiple computations in the cache, which has much faster access to the processor than main memory.

This will have both nice notation and efficiency *for this expression*. What about a group of expressions? For example, consider inverting a symmetric, 3×3 matrix (rank 2 tensor) `A`. Because it is small, a fairly good method is to do it directly

```
det=A(0,0)*A(1,1)*A(2,2) + A(1,0)*A(2,1)*A(0,2)
    + A(2,0)*A(0,1)*A(1,2) - A(0,0)*A(2,1)*A(1,2)
    - A(1,0)*A(0,1)*A(2,2) - A(2,0)*A(1,1)*A(0,2);
I(0,0)= (A(1,1)*A(2,2) - A(1,2)*A(1,2))/det;
I(0,1)= (A(0,2)*A(1,2) - A(0,1)*A(2,2))/det;
I(0,2)= (A(0,1)*A(1,2) - A(0,2)*A(1,1))/det;
I(1,1)= (A(0,0)*A(2,2) - A(0,2)*A(0,2))/det;
I(1,2)= (A(0,2)*A(0,1) - A(0,0)*A(1,2))/det;
I(2,2)= (A(1,1)*A(0,0) - A(1,0)*A(1,0))/det;
```

Through the magic of expression templates, this will then get transformed into something like

```
for(int n=0;n<N;++n)
  det[n]=A[0][0][n]*A[1][1][n]
         * A[2][2][n]
         + A[1][0][n]*A[2][1][n]
         * A[0][2][n]
         + A[2][0][n]*A[0][1][n]
         * A[1][2][n]
         - A[0][0][n]*A[2][1][n]
         * A[1][2][n]
         - A[1][0][n]*A[0][1][n]
         * A[2][2][n]
         - A[2][0][n]*A[1][1][n]
         * A[0][2][n];
for(int n=0;n<N;++n)
  I[0][0][n]= (A[1][1][n]*A[2][2][n]
             - A[1][2][n]*A[1][2][n])
             / det[n];
for(int n=0;n<N;++n)
  I[0][1][n]= (A[0][2][n]*A[1][2][n]
             - A[0][1][n]*A[2][2][n])
             / det[n];
for(int n=0;n<N;++n)
  I[0][2][n]= (A[0][1][n]*A[1][2][n]
             - A[0][2][n]*A[1][1][n])
             / det[n];
for(int n=0;n<N;++n)
  I[1][1][n]= (A[0][0][n]*A[2][2][n]
             - A[0][2][n]*A[0][2][n])
             / det[n];
for(int n=0;n<N;++n)
  I[1][2][n]= (A[0][2][n]*A[0][1][n]
             - A[0][0][n]*A[1][2][n])
             / det[n];
for(int n=0;n<N;++n)
  I[2][2][n]= (A[1][1][n]*A[0][0][n]
             - A[1][0][n]*A[1][0][n])
             / det[n];
```

Once again, we have multiple loops over the grid of N points. We also have a temporary, `det`, which will be moved between the processor and memory multiple times and can not be saved in the cache. In addition, each of the elements of `A` will get transferred four times. If we instead manually fuse the loops together

```
for(int n=0;n<N;++n)
  {
    double det=A[0][0][n]*A[1][1][n]
               * A[2][2][n]
               + A[1][0][n]*A[2][1][n]
               * A[0][2][n]
               + A[2][0][n]*A[0][1][n]
               * A[1][2][n]
               - A[0][0][n]*A[2][1][n]
               * A[1][2][n]
               - A[1][0][n]*A[0][1][n]
               * A[2][2][n]
               - A[2][0][n]*A[1][1][n]
               * A[0][2][n];
    I[0][0][n]=(A[1][1][n]*A[2][2][n]
                - A[1][2][n]*A[1][2][n])/det;
    // and so on for the other
     indices.
    .
    .
    .
  }
```

then `det` and the elements of `A` at a particular `n` can fit in the cache while computing all six elements of `I`. After that, they won't be needed again. For $N = 100,000$ this code takes anywhere from 10% to 50% less time (depending on architecture) while using less memory. This is not an isolated case. In General Relativity

codes, there can be over 100 named temporaries like `det`. Unless the compiler is omniscient, it will have a hard time fusing all of the loops between statements and removing extraneous temporaries. It becomes even more difficult if there is an additional loop on the outside which loops over multiple grids, as is common when writing codes that deal with multiple processors or adaptive grids.

As an aside, the Blitz library [11] uses this approach. On the benchmark page for the Origin 2000/SGI C++ [12], there are results for a number of loop kernels. For many of them, Blitz compares quite favorably with the Fortran versions. However, whenever there is more than one expression with terms common to both expressions (as in loop tests #12–14, 16, 23–24) there are dramatic slow downs. It even mentions explicitly (after loop test #14) "The lack of loop fusion really hurts the C++ versions."

The POOMA library [9] uses an approach which should solve some of these problems. It splits up calculations into chunks that fit neatly into the cache. Then, multiple loops are no longer a problem, because all of the variables in each loop fit into the cache. Doing this correctly is quite difficult, because the library writer must be able to deduce how many different variables are involved in each execution block. In addition, it still requires storage for named temporaries.

Does all this mean that we have to go back to C-tran for performance?

2.3. *Expression templates + manual loop fusion*

The flaw in the previous method is that it tried to do two things at once: implicitly sum indices and iterate over the grid. Iterating over the grid while inside the expression necessarily meant excluding other expressions from that iteration. It also required temporaries to be defined over the entire grid. To fix this, we need to manually fuse all of the loops, and provide for temporaries that won't be defined over the entire grid. We do this by making two kinds of tensors. One of them just holds the elements (so a Tensor1 would have three doubles, and a Tensor2 has 9 doubles). This is used for the local named temporaries. The other kind holds pointers to arrays of the elements. To iterate over the array, we overload `operator++`. A rough sketch of this tensor iterator class is

```
class Tensor1_iter
  {
     mutable double *x, *y, *z;
   public:
     void operator++()
       {
          ++x;
          ++y;
          ++z;
       \
     \\Indexing, assignment, initia-lization
      operators etc.
  }
```

Making it a simple `double *` allows us to use any sort of contiguous storage format for the actual data. The data may be managed by other libraries, giving us access to a pointer that may change. In that sense, the Tensor is not the only owner of the data, and all copies of the Tensor have equal rights to access and modify the data.

We make the pointers `mutable` so that we can iterate over `const Tensor1_iter`'s. The indexing operators for `const Tensor1_iter` returns a `double`, not `double *` or `double &`, so the actual data can't be changed. This is different from how iterators in the Standard Template Library work. This keeps the data logically `const`, while allowing us to look at all of the points on the grid for that `const Tensor1_iter`.

We would then write the matrix inversion example as

```
for(int n=0;n<N;++n)
  {
    double det=A(0,0)*A(1,1)*A(2,2) + A(1,0)
               * A(2,1)*A(0,2)
               + A(2,0)*A(0,1)*A(1,2) - A(0,0)
               * A(2,1)*A(1,2)
               - A(1,0)*A(0,1)*A(2,2) - A(2,0)
               * A(1,1)*A(0,2);
    I(0,0)= (A(1,1)*A(2,2) - A(1,2)*A(1,2))/det;
    I(0,1)= (A(0,2)*A(1,2) - A(0,1)*A(2,2))/det;
    I(0,2)= (A(0,1)*A(1,2) - A(0,2)*A(1,1))/det;
    I(1,1)= (A(0,0)*A(2,2) - A(0,2)*A(0,2))/det;
    I(1,2)= (A(0,2)*A(0,1) - A(0,0)*A(1,2))/det;
    I(2,2)= (A(1,1)*A(0,0) - A(1,0)*A(1,0))/det;
    ++I;
    ++A;
  }
```

This solution is not ideal and has a few hidden traps, but is certainly better than C-tran. It requires a manually created loop over the grid, and all relevant variables have to be incremented. Care must also be taken not to iterate through a grid twice. For example, following the above code fragment with

```
for(int n=0;n<N;++n)
  {
    Trace=A(0,0)+A(1,1)+A(2,2);
    ++A;
    ++Trace;
  }
```

will iterate off of the end of A, because the internal pointer for A has not been reset.

For our specific application (Numerical General Relativity), these were not serious problems, because most of the logic of our program is in the manipulation of local named variables. Only a few variables (the input and output) need to be explicitly iterated.

However, this may not be the right kind of solution for generic arrays. They correspond to rank 0 tensors (tensors without any indices). It is a win for higher rank tensors because most of the complexity is in the indices. But for generic arrays, there are no indices. A solution like this would look almost identical to C-tran.

3. Implementing natural notation efficiently

The previous section described the design choices required for efficiency. In this section, we describe how we achieve a natural tensor notation with minimal overhead.

3.1. Basic classes

To illustrate our basic design, we start with rank 1 tensors. The code is slightly more clear for `Tensor1`'s than for `Tensor1_iter`'s, so we will concentrate on them. However, the techniques are the same, and almost all of the code is used by both types.

We define a class `Tensor1` with three elements corresponding to the x, y, and z components. We define `operator()(int)` to return these elements, so if we have a `Tensor1` named A, `A(0)` gives the x element, `A(1)` gives the y element, and `A(2)` gives the z element. The outline of this class so far is

```
class Tensor1
{
  double data0, data1, data2;
public:
  double & operator(int N)
    {
      return (N==0 ? data0 : (N==1 ? data1 :
              data2));
    }
.
.
.
```

Note that there is no range check on the index, so `A(1221)` will return the same result as `A(2)`. We also have a checked version selected at compile time with `#ifdef DEBUG` macros, but we omit it for clarity.

We want to support the notation `A(i) = B(i)`, where `i` is implicitly summed over 0, 1, and 2. To do this, we use expression templates [13], because they transparently provide high performance. We define two auxiliary classes, `Index` and `Tensor1_Expr`. `Index` is used to tell the compiler what the index of the `Tensor1` is. It uses a template parameter to store this information, so it is otherwise empty. The definition of `Index` is thus rather simple

```
template<char i>
class Index {};
```

On the other hand, `Tensor1_Expr` is designed to hold any kind of expression that eventually simplifies to a rank 1 tensor. For example, the expressions `A(i)` and `B(j)*T(j,i)` (which has an implicit summation over `j`) both simplify to a tensor with one index. To accomplish this, `Tensor1_Expr` has two template parameters that tell it 1) what kind of object it contains, and 2) what its index is. This class is analogous to the DExpr class in the original expression templates paper [13]. The definition for `Tensor1_Expr` is then

```
template<class A, char i>
class Tensor1_Expr
{
  A iter;
public:
  Tensor1_Expr(A &a): iter(a) {}
  double operator()(const int N) const
    {
      return iter(N);
    }
};
```

Here, the template parameter `class A` is the object contained in the expression, and `char i` is the tensor index. We overload the member function `Tensor1::operator()(Index)` to return a `Tensor1_Expr`.

```
template<char i>
Tensor1_Expr<Tensor1,i> operator() (Index<i>
    index)
  {
     return Tensor1_Expr<Tensor1,i>(*this);
  }
```

An example of its use is

```
Index<'i'> i;
Tensor1 A;
A(i);
```

Here, the statement `A(i);` creates a `Tensor1_Expr<Tensor1,'i'>`. This just illustrates the simplest case, where a `Tensor1_Expr` holds a `Tensor1`. To assign one tensor to another, we create a partial specialization of `Tensor1_Expr` for the case when it contains a `Tensor1`

```
template<char i>
class Tensor1_Expr<Tensor1, i>
{
  Tensor1 &iter;
public:
  Tensor1_Expr(Tensor1 &a): iter(a) {}
  double & operator()(const int N)
    {
      return iter(N);
    }
  template<class B>
  const Tensor1_Expr<Tensor1,i> &
    operator=(const Tensor1_Expr<B,i>&result)
      {
        iter(0)=result(0);
        iter(1)=result(1);
        iter(2)=result(2);
        return *this;
      }
  const Tensor1_Expr<Tensor1,i> &
      operator=(const Tensor1_Expr
                 <Tensor1,i> &result)
      {
        return operator=<Tensor1>(result);
      }
};
```

This is almost the same as the general `Tensor1_Expr`. The only differences are that it defines the equals operator, and it takes a reference to the object that it contains (`Tensor1 &iter`), instead of a copy (`A iter`). The second change is needed in order for assignment to work. Our example now becomes

```
Index<'i'> i;
Tensor1 A, B;
A(i)=B(i);
```

The last statement creates two `Tensor1_Expr<Tensor1,'i'>`'s, one for `A` and one for `B`. It then assigns the elements of `B` to the elements of A. If we had tried something like

```
Index<'i'> i;
Index<'j'> j;
Tensor1 A, B;
A(i)=B(j);
```

then the compiler would not have found a suitable `operator=()`. The second `Tensor1_Expr<>` template parameter (`char`), which was obtained from `Tensor1::operator()(Index<i> index)`, would not match. This provides strong compile-time checking of tensor expressions.

Generalizing this to higher rank tensors is fairly straightforward. We define the appropriate `TensorN` class to hold more elements (3^N). We overload `operator()(int,int,...)` and `operator()(Index,Index,...)`. We define a `TensorN_Expr<>` class and overload its `operator()(int,int,...)`. We partially specialize it for `TensorN`'s and define an equal's operator.

3.2. *Arithmetic operators*

Now we want to do something really useful. We want to add two `Tensor1`'s together. This is where expression templates really come into play. We do this by creating a helper class `Tensor1_plus_Tensor1`. In the original expression templates paper[13], there was a generalized DBinOpExpr class. However, the action of * is very different from $+$ and $-$, and $/$ is not well defined for tensors of rank 1 or higher, so there is no real advantage to a generalized BinOpExpr class. The helper class is defined as

```
template<class A, class B, char i>
class Tensor1_plus_Tensor1
{
  const Tensor1_Expr<A,i> iterA;
  const Tensor1_Expr<B,i> iterB;
public:
  double operator()(const int N) const
    {
      return iterA(N)+iterB(N);
    }
Tensor1_plus_Tensor1(const Tensor1_Expr<A,i>
                     &a,
                     const Tensor1_Expr<B,i>
                     &b): iterA(a),
                     iterB(b) {}
};
```

This helper class contains the two objects that are being added. When we use `operator()(int)` to ask for an element, it returns the sum of the two objects. This class is used in the definition of `operator+(Tensor1_Expr,Tensor1_Expr)`

```
template<class A, class B, char i>
inline Tensor1_Expr<const Tensor1_plus_Tensor1
  <const Tensor1_Expr<A,i>,const
  Tensor1_Expr<B,i>,i>,i>
operator+(const Tensor1_Expr<A,i>&a,
          const Tensor1_Expr<B,i>&b)
  {
      typedef const Tensor1_plus_Tensor1
      <const Tensor1_Expr<A,i>,
      const Tensor1_Expr<B,i>,i>TensorExpr;
    return Tensor1_Expr<TensorExpr,i>
    (TensorExpr(a,b));
  }
```

Note that the indices of the two `Tensor1_Expr`'s have to match up, or they won't have the same `char` template parameter. This is another example of strict compile-time checking for validity of tensor expressions.

To make more sense of this, let's consider an example

```
Index<'i'> i;
Tensor1 A, B, C;
A(i)=B(i)+C(i);
```

The individual expressions `A(i)`, `B(i)` and `C(i)` all create a `Tensor1_Expr<Tensor1,'i'>`. The plus operator creates a `Tensor1_Expr<Tensor1_plus_Tensor1<Tensor1,Tensor1,'i'>,'i'>`. The equals operator then asks for `operator()(0)`, `operator()(1)`, and `operator()(2)` from this compound object. The `Tensor1_Expr<>` object passes these calls to it's contained object, the `Tensor1_plus_Tensor1`. The `Tensor1_plus_Tensor1` object returns the sum of the calls to the two objects (`Tensor1_Expr`'s) it contains. The `Tensor1_Expr`'s pass the call to the `Tensor1` it contains, and we finally get the results.

The code for subtraction is exactly the same with $+$ replaced with $-$ and `_plus_` replaced with `_minus_`. The `*` operator has a very different meaning which depends on what the indices are. For example, `A(i)*B(i)` contracts the two tensors together, implicitly summing the indices, yielding a `double`, while `A(i)*B(j)` creates a new `Tensor2` with indices of `i` and `j`. Implicit summation is described in the next section, but the solution to the latter is quite similar to the addition operator described before. We first need a helper class

```
template<class A, class B, char i, char j>
class Tensor1_times_Tensor1
{
  const Tensor1_Expr<A,i> iterA;
  const Tensor1_Expr<B,j> iterB;
public:
  Tensor1_times_Tensor1(const Tensor1_Expr<A,
                        i> &a,
                        const Tensor1_Expr
                        <B,j> &b)
    : iterA(a), iterB(b) {}
  double operator()(const int N1,const int N2)
                    const
  {
    return iterA(N1)*iterB(N2);
  }
};
```

and then we overload `operator*(Tensor1_Expr,Tensor1_Expr)`

```
template<class A, class B, char i,
char j> inline
Tensor2_Expr<const Tensor1_times_Tensor1
              <const Tensor1_Expr<A,i>,
               const Tensor1_Expr<B,j>,i,j>,
               i,j>
  operator*(const Tensor1_Expr<A,i> &a,
            const Tensor1_Expr<B,j> &b)
{
  typedef const Tensor1_times_Tensor1
    <const Tensor1_Expr<A,i>,
     const Tensor1_Expr<B,j>,i,j> TensorExpr;
  return Tensor2_Expr<TensorExpr,i,j>
  (TensorExpr(a,b));
}
```

3.3. Implicit summation

The preceding work is not really that interesting. Blitz [11] already implements something almost like this. What really distinguishes this library from others is its natural notation for implicit summation, or contraction. There are two kinds of contraction: external and internal.

3.3.1. External contraction

External contraction is when the index of one tensor contracts with the index of another tensor. This is the most common case. Consider the simple contraction of two rank 1 tensors

```
Index<'i'> i;
Tensor1 A,B;
double result=A(i)*B(i);
```

To accomplish this, we specialize `operator*(Tensor1_Expr,Tensor1_Expr)`

```
template<class A, class B, char i>
inline double operator*(const Tensor1_Expr
                         <A,i> &a,
                        const Tensor1_Expr<B,
                        i>&b)
{
  return a(0)*b(0) + a(1)*b(1) + a(2)*b(2);
}
```

Because the function is typed on the template parameter `i`, which comes from the `Index` when the `Tensor1_Expr` is created, it will only be called for operands that have the same index (i.e. `A(i)*B(i)`, not `A(i)*B(j)`).

We also want to contract tensors together that result in a tensor expression, such as a `Tensor1` contracted with a `Tensor2` (`A(i)*T(i,j)`). As with the addition and subtraction operators, we use a helper class

```
template<class A, class B, char i,char j>
class Tensor2_times_Tensor1_0
{
  const Tensor2_Expr<A,j,i> iterA;
    const Tensor1_Expr<B,j> iterB;
public:
  Tensor2_times_Tensor1_0(const Tensor2_Expr
                          <A,j,i> &a,
                          const Tensor1_Expr
                          <B,j> &b)
    : iterA(a), iterB(b) {}
  double operator()(const int N1) const
    {
      return iterA(0,N1)*iterB(0)
              + iterA(1,N1)*iterB(1)
              + iterA(2,N1)*iterB
              (2);
    }
};
```

The _0 appended to the end of the class is a simple way of naming the classes, since we will need a similar class for the case of `A(i)*T(j,i)` (as opposed to `A(i)*T(i,j)`, which we have here). Then we specialize `operator*(Tensor1_Expr,Tensor2_Expr)`

```
template<class A, class B, char i, char j>
        inline
Tensor1_Expr<const Tensor2_times_Tensor1_1
              <const Tensor2_Expr<A,i,j>,
               const Tensor1_Expr<B,j>,i,
               j>,i>
  operator*(const Tensor1_Expr<B,j> &b,
            const Tensor2_Expr<A,i,j> &a)
{
  typedef const Tensor2_times_Tensor1_1
    <const Tensor2_Expr<A,i,j>,
     const Tensor1_Expr<B,j>,i,j> TensorExpr;
  return Tensor1_Expr<TensorExpr,i>
  (TensorExpr(a,b));
}
```

3.3.2. *Internal contraction*

Contraction can also occur within a single tensor. The only requirement is that there are two indices to contract against each other. A simple example would be

```
Index<'i'> i;
Tensor2 T;
double result=T(i,i);
```

The last line is equivalent to

```
double result=T(0,0)+T(1,1)+T(2,2);
```

This internal contraction is simply implemented by specializing `Tensor2::operator()(Index,Index)`

```
template<char i>
double operator()(const Index<i> ind-ex1,
const Index<i> index2)
{
  return data00 + data11 + data22;
}
```

There is also a more complicated case where there is an internal contraction, but the result is still a tensor. For example, a rank 3 tensor `W` contracting to a rank 1 `W(i,j,j)`. For this, we define a helper class

```
template<class A, char i>
class Tensor3_contracted_12
{
  const A iterA;
public:
  double operator()(const int N) const
  {
    return iterA(N,0,0) + iterA(N,1,1) + iterA
    (N,2,2);
  }
  Tensor3_contracted_12(const A &a):  iterA(a)
  {}
};
```

Then we define a specialization of `operator()(Index,Index,Index)` to create one of these objects

```
template<char i, char j> inline
Tensor1_Expr<const Tensor3_contracted_12
<Tensor3_dg,i>,i>
operator()(const Index<i> index1,
           const Index<j> index2,
           const Index<j> index3) const
{
  typedef const Tensor3_contracted_12
  <Tensor3_dg,i> TensorExpr;
  return Tensor1_Expr<TensorExpr,i>
  (TensorExpr(*this));
}
```

Now, if we ask for the x component of `W(i,j,j)`, the compiler will automatically sum over the second and third indices, returning `W(0,0,0)+W(0,1,1)+W(0,2,2)`.

3.4. *Reduced rank tensors*

Expressions like `A(i)=T(0,i)` can sometimes pop up. To handle this, we make a helper class

```
template<class A>
class Tensor2_numeral_0
{
  A iterA;
  const int N;
public:
  double operator()(const int N1) const
  {
    return iterA(N,N1);
  }
Tensor2_numeral_0(A &a, const intNN):iterA(a),
N(NN) {}
};
```

This class is instantiated when `operator()(Index<>, int)` is called on a `Tensor2`

```
template<char i>
Tensor1_Expr<const Tensor2_numeral_0<const
Tensor2>,i> operator()(constint int N,
{
  typedef const Tensor2_numeral_0<const
  Tensor2> TensorExpr;
  return Tensor1_Expr<TensorExpr,i>
  (TensorExpr(*this,N));
}
```

The end result of all of this is that when we write

```
Index<'i'> i;
Tensor1 A;
Tensor2 T;
A(i)=T(0,i);
```

we create a `Tensor1_Expr<Tensor2_numeral_0 <Tensor2>,'i'>` which then gets assigned to the `Tensor1_Expr<Tensor1,'i'>` created by `A(i)`.

Unfortunately, a syntax like `T(0,i)` is inefficient, because the value of the first index (0) is difficult, even in simple cases, for compilers to deduce and apply at compile time. To aid the compiler in the case where the programmer does know at compile-time what the value of the index is, we introduce a new auxiliary class

```
template<int N>
class Number
{
public:
  Number() {};
  operator int() const
  {
    return N;
  }
};
```

Like `Index`, there is very little to the actual class. Because of the conversion operator, it can be used anywhere an `int` can be used. For example

```
Number<1> N1;
Tensor1 A,B;
A(N1)=B(N1);
```

is equivalent to

```
Tensor1 A,B;
A(1)=B(1);
```

We also create another helper class

```
template<class A, char i, int N>
class Tensor2_number_0
{
  const A &iterA;
public:
  double & operator()(const int N1)
  {
    return iterA(N,N1);
  }
  double operator()(const int N1) const
  {
    return iterA(N,N1);
  }
  Tensor2_number_0(A &a): iterA(a) {}
};
```

As with `int`'s, this class is instantiated when `operator()(Number<>,Index<>)` is called on a `Tensor2`

```
template<char i, int N>
Tensor1_Expr<const Tensor2_number_0<const
Tensor2,i,N>,i>
operator()(const Number<N> n1, const Index<i>
index1) const
{
  typedef const Tensor2_number_0<const
  Tensor2,i,N> TensorExpr;
  return Tensor1_Expr<TensorExpr,i>
  (TensorExpr(*this));
}
```

An example of usage is

```
Number<0> N0;
Index<'i'> i;
Tensor1 A;
Tensor2 T;
A(i)=T(N0,i);
```

Simple tests show that a good compiler can use the template arguments to optimize this expression as well as if it were written with simple arrays, while it can't optimize expressions with simple `int`'s as indices.

3.5. Symmetric/antisymmetric tensors

It is often the case that a tensor will have various symmetries or antisymmetries, such as `S(i,j)`=`S(j,i)`, or `A(i,j)`= $-$`A(j,i)`. Taking advantage of these symmetries can significantly reduce storage and computation requirements. For example, a symmetric rank 2 tensor `S` only has 6 truly independent elements (`S(0,0)`, `S(0,1)`, `S(0,2)`, `S(1,1)`, `S(1,2)`, `S(2,2)`), instead of 9. The other three elements (`S(1,0)`, `S(2,0)`, `S(2,1)`) are simply related to the previous elements. An antisymmetric rank 2 tensor `A` only has 3 independent elements (`A(0,1)`, `A(0,2)`, `A(1,2)`). Three of the other elements are simply related to these three (`A(1,0)`= $-A$`(0,1)`, `A(2,0)`= $-$`A(0,2)`, `A(2,1)`= $-A$`(1,2)`). The rest (`A(0,0)`, `A(1,1)`, and `A(2,2)`) must be zero, since `A(0,0)`=-`A(0,0)` etc. The effect becomes more dramatic with higher rank tensors. The Riemann tensor mentioned before has four indices, making a total of 81 possible elements, but symmetries and antisymmetries reduce that number to 6.

3.5.1. Symmetric tensors

It turns out that implementing a symmetric tensor is quite simple. First, we define a class (e.g. `Tensor2_symmetric`) with the minimum number of elements. Then we write the indexing operators (`operator()(int,int,...)`) so that, if an element that is not available is requested, it uses the symme-

try and returns the equivalent one. For example, for a symmetric rank 2 tensor, we only define `data00`, `data01`, `data02`, `data11`, `data12`, and `data22`. Then, if element `(2,1)` is requested, we just return `data12`.

Second, we simplify the equals operator so that it only sets the elements we have. That is, for a normal `Tensor2`, we would have

```
iter(0,0)=result(0,0);
iter(0,1)=result(0,1);
iter(0,2)=result(0,2);
iter(1,0)=result(1,0);
iter(1,1)=result(1,1);
iter(1,2)=result(1,2);
iter(2,0)=result(2,0);
iter(2,1)=result(2,1);
iter(2,2)=result(2,2);
```

while for a `Tensor2_symmetric` we only have

```
iter(0,0)=result(0,0);
iter(0,1)=result(0,1);
iter(0,2)=result(0,2);
iter(1,1)=result(1,1);
iter(1,2)=result(1,2);
iter(2,2)=result(2,2);
```

We also have to write all of the arithmetic and contraction operators that use `Tensor2_symmetric`'s, but they are basically the same as the no-symmetry case.

3.5.2. Antisymmetric tensors

Implementing antisymmetric tensors is a bit more tricky. The same kinds of changes are made to the definitions of `Tensor` and `Tensor_Expr`, but it is not clear what should be returned when an `operator()(int,int,...)` asks for an element that is identically zero (such as `A(0,0)`) or is the negative of the value that we store (such as `A(1,0)`). The imperfect solution that we have is to rename `T& operator()(int,int,...)` to `T& unsafe(int,int,...)`. However, `T operator()(int,int,...) const` is still defined, and returns the appropriate value. This allows a library user to access the elements of the antisymmetric tensor using the natural syntax, but assigning to it will be obviously unsafe. However, if only `Index`'s are used, then the library will automatically sum over the correct elements. It uses `unsafe` internally, so most of the time, the library user will not even need to know about `unsafe`. In those cases where the user does have to use `unsafe`, debug builds should catch any illegal assignment during runtime.

4. Implementation and testing

We have implemented the (inefficient) "Simple Classes" method (Section 2.1) and the (efficient) "Expression Templates + Manual Loop Fusion" method (Section 2.3). We did not attempt to implement "Expression Templates" (Section 2.2), because it was clear that it could not be as efficient as the method with manual loop fusion, while still being a difficult chore to implement.

4.1. Compiler compatibility

Not all compilers support enough of the C++ standard to compile expression templates, while simple classes work with almost any compiler. A comparison of twelve combinations of compiler and operating system is shown in Table 1.

The template support of these compilers is actually quite good. A year ago many of these compilers would not have been able to compile the efficient library. We tested a slightly out of date version of SGI's compiler, with the current version being 7.3.1.3. Even so, it's only problem is that it does not make <cmath> available, and it is easy to work around that. IBM's compiler seems to be immature, with a remarkable number of cases of Internal Compiler Errors (ICE's). They are currently in beta testing for version 6.0 which may rectify that. The Sun compiler is the only one that was completely unable to compile the efficient library. They have also come out with a new version recently, which may fix those problems.

The C++ standard specifies that compliant programs can only rely on 17 levels of template instantiation. Otherwise, it would be difficult to detect and prevent infinite recursion. However, the intermediate types produced by template expression techniques can exceed this limit. Most compilers allowed us to override the limit on the number of pending instantiations, with the exception of the SGI, Portland Group, and IBM compilers (although the Intel command line option, "-Qoption,cpp,–pending_instantiations,N", is not documented). The SGI and Portland group compilers would not compile any program with too many levels. The IBM compiler did not honor the standard and happily compiled programs with more than 50 levels of template instantiation.

This is not a complete list of C++ compilers. Notably, it does not include the Microsoft, Borland, Metrowerks, or HP compilers. The Microsoft compiler probably can not compile the efficient library since

Table 1
Compilers comparison

Compiler/Operating System	Compiles efficient library?
Comeau como 4.2.45.2 + libcomobeta14/Linux x86 [15] Compaq cxx 6.3/Tru64 [16] GNU gcc 3.1/Linux x86, Solaris, AIX [17] KAI KCC 4.0d/Linux x86, 3.4/AIX [18] Intel icc 6.0-149/Linux x86 [21] Portland Group pgCC 4.0-1/Linux x86 [22]	Yes
IBM xlC 5.0.0.1/AIX [19]	Yes- with occasional ICE's
SGI CC 7.3.1.1m/Irix [20]	Somewhat-no <cmath> and can't override template instantiation limit
Sun CC 6.2/Solaris Sparc [23]	No, doesn't support partial specialization with non-type template parameters

it does not implement partial specialization [24]. On Linux, the Metrowerks compiler is actually the GNU gcc 2.95.2 compiler with a fancy GUI. That version of gcc will compile this library, but it can't compile the more general version that will be described in Section 5. As for Metrowerks on other platforms, as well as the Borland and HP compilers, it is impossible to say anything without actually trying them out. This template library tends to flush out obscure bugs in compilers that claim full compliance. It is useful enough in that regard to become part of the official release criteria for GNU gcc [25], in addition to the POOMA [9] and Blitz [11] template libraries.

4.2. Benchmarks

We have three tests to measure the efficiency of the various methods relative to each other and to C-tran. We have not attempted a direct comparison with other tensor libraries, because most do not support implicit summation and none of them support the wide range of tensor types needed (ranks 1, 2, 3 and 4 with various symmetries). This makes replicating the functionality in the tests extremely time consuming.

4.2.1. Small loop kernel: Implementation test

To make sure that we didn't make any gross errors in implementing expression templates, we use a small loop kernel to compare the efficient library against C-tran style. The tensor version is simply

```
Tensor1 x(0,1,2), y(3,4,5), z(6,7,8);
for(int n=0;n<1000000;n++)
  {
    Index<'i'> i;
    x(i)+=y(i)+z(i);
      +(y(i)+z(i))-(y(i)+z(i))
      +(y(i)+z(i))-(y(i)+z(i))
      +(y(i)+z(i))-(y(i)+z(i))
        .
        .
        .
  }
```

The complexity of the expression is determined by how many (y(i)+z(i))−(y(i)+z(i)) terms there are in the final expression. Note that since we're adding and subtracting the same amounts, the essential computation has not changed. We also coded a version of the code in C-tran style using ordinary arrays, and compared the execution speeds of the two versions.

For large expressions, KCC was the only compiler that could fully optimize away the overhead from the expression templates, although we had to turn off exceptions. This is good evidence that we didn't make any serious optimization errors in implementation. For the other compilers, the slowdown increased with the number of expressions, becoming more than 100 times slower than the C-tran version.

This benchmark may be deceiving, though. The C-tran versions all run at the same speed regardless of how many terms we added. An examination of the assembler output shows that the compiler removes the identically zero subexpressions. This wouldn't be possible in most production codes, so the relative slowdown may not be as great.

4.2.2. Small loop kernel: Compiler optimization test

To get a better handle on how well the compilers optimize more realistic code, we wrote several versions of a code to compute an infinite sum

$$a1_i \sum_{n=0}^{\infty} 0.1^n + 2 \cdot a2_i \sum_{n=0}^{\infty} 0.02^n + 3\left(a1_j a2^j\right)$$

$$a3_i \sum_{n=0}^{\infty} 0.006^n + \ldots .$$

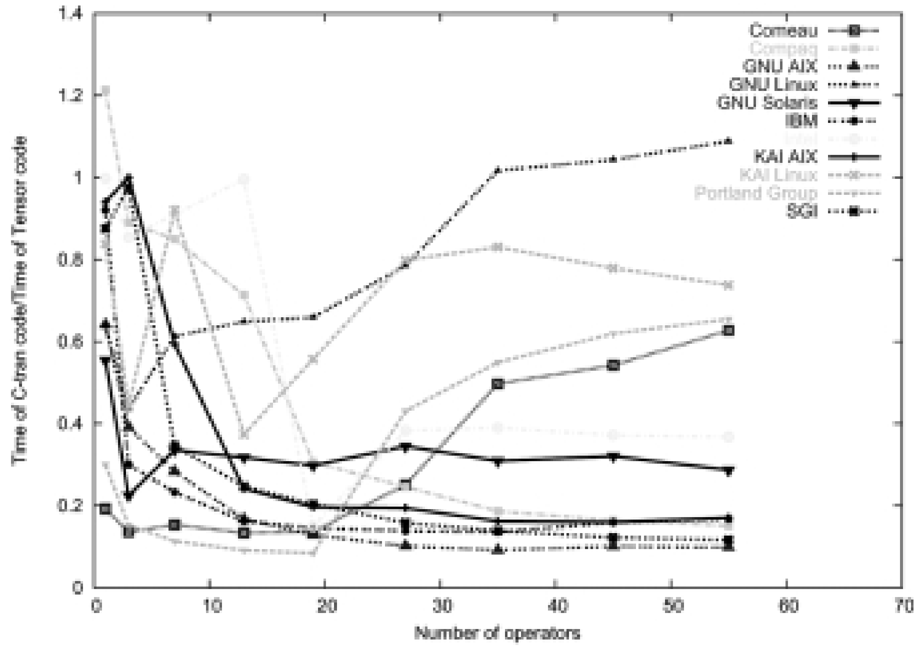


Fig. 1. Relative execution times of C-tran and Tensor1's. Any points less than one mean that the Tensor code is slower than C-tran.

The simplest version computes the sum that only has the $(a1_i)$ term, the second version has both the $(a1_i)$ and the $(2 \cdot a2_i)$ terms, and so on. This gives the compiler a more and more complicated expression to optimize. The code for the tensor version is then

```
Tensor1 y(0,1,2);
Tensor1 a1(2,3,4);
Tensor1 a2(5,6,7);
Tensor1 a3(8,9,10);
Tensor1 a4(11,12,13);
Tensor1 a5(14,15,16);
.
.
.
for(int n=0;n<1000000;++n)
  {
    const Index<‘i’> i;
    const Index<‘j’> j;
    const Index<‘k’> k;
    const Index<‘l’> l;
    const Index<‘m’> m;
    y(i)+=a1(i)
          + 2*a2(i)
          + 3*a1(j)*a2(j)*a3(i)
          + 4*a1(j)*a3(j)*a2(k)*a2(k)*a4(i)
          + 5*a1(j)*a4(j)*a2(k)*a3(k)*a5(i)
          .
          .
          .
    a1(i)*=0.1;
    a2(i)*=0.2;
    a3(i)*=0.3;
    a4(i)*=0.4;
    a5(i)*=0.5;
    .
    .
    .
  }
```

with complexity determined by how much we fill in the ellipses. After n gets to about fifteen, the sum converges to machine precision, although current compilers can not deduce that. We vary the number of iterations (1000000 here), so that the run finishes in a reasonable time. We also laboriously coded a C-tran version and compared the execution speed of the two. Figure 1 plots the relative execution times of the the Tensor1 and C-tran versions versus the number of operators (+ and *) in the expressions.

The specific compiler options used to create this plot are listed in the appendix. The performance of some of the compilers may be a little overstated since they don't optimize the C-tran code as well as some other compilers. On Linux x86, the fastest compiler for the C-tran code was either the Intel or KAI compiler, and on AIX, it was IBM's xlC. So in Figs 2 and 3 we plot the relative execution time of the fastest C-tran codes versus the various Tensor codes for Linux and AIX.

These are busy plots, but the main thing to notice is that sometimes the compilers can optimize the expressions well, though often not. Some compilers do well with small expressions, and some do better with larger

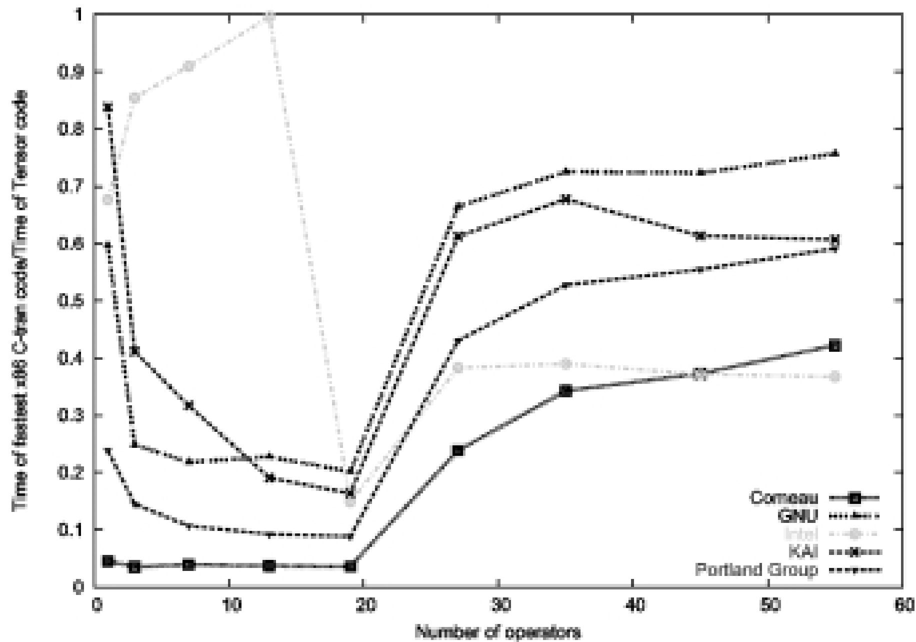


Fig. 2. Relative execution times of fastest C-tran and Tensor's on Linux x86.

expressions. However, even for the best compilers, the Tensor1 class can run much slower than C-tran. In particular, the non-x86 compilers seem to fare worse than their x86 cousins.

4.2.3. Complete application: Numerical relativity

To gauge how much the lack of optimization matters in real codes, we used our application for simulating neutron star collisions in General Relativity [5]. This code has to compute many complicated formulas on grids much too large to fit into processor caches. We found that, when compiled with KAI's KCC 4.0d compiler on an IBM SP2 running AIX, the "Expression Templates+Manual Loop Fusion" library (Section 2.3) runs about twice as fast and uses a third of the memory of the "Simple Classes" library (Section 2.1). This was after we attempted to optimize the "Simple Classes" code by manually constructing expression trees to reduce temporaries [26]. Unfortunately, the expression trees quickly became too numerous and varied to create by hand.

When compiled with GNU gcc 2.95.2 or IBM's xlC 5.0.1.0, the "Expression Templates+Manual Loop Fusion" library code was 10–20% slower than when compiled with KCC. The logic was far too complicated to create a C-tran version.

Because these tests were run with different compiler versions on different machines than that used for the loop kernels, we plot the results analogous to Fig. 3 in Fig. 4. KAI's compiler does well with small expressions, which may imply that the application spends more time in small tensor expressions. It is difficult to tell for sure, because the compiler inlines almost everything, making direct measurement tricky. But in that case, the difference between KCC and gcc would be much larger. This suggests that the differences lie elsewhere, and the bottleneck is not expression templates. It is impossible to say anything for sure.

5. Extending the library

An experienced reader may have looked at the rough declaration of `Tensor1` and thought that hard coding it to be made up of `double` is rather short sighted. It is not so difficult to envision the need for tensors made up of `int`'s or `complex<double>`'s. It might also be nice to use two or four dimensional tensors (so a `Tensor1` would have 2 or 4 elements, a `Tensor2` would have 4 or 16 elements). The obvious answer is to make the type and dimension into template parameters. We can use arrays for the elements, in which case the class becomes

```
template<class˜T, int Dim> class Ten
sor1 {
  T data[Dim];
```

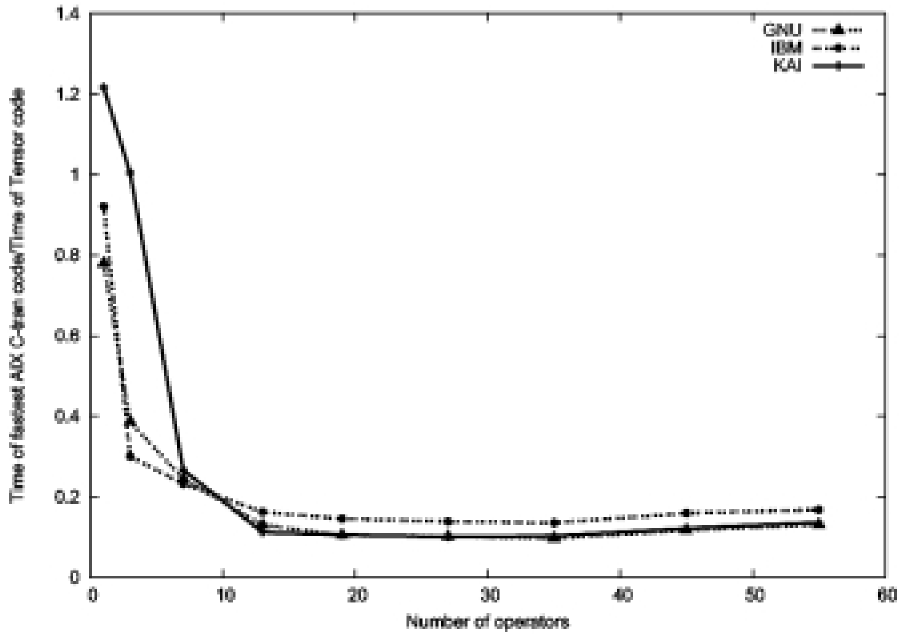


Fig. 3. Relative execution times of fastest C-tran and Tensor's on AIX.

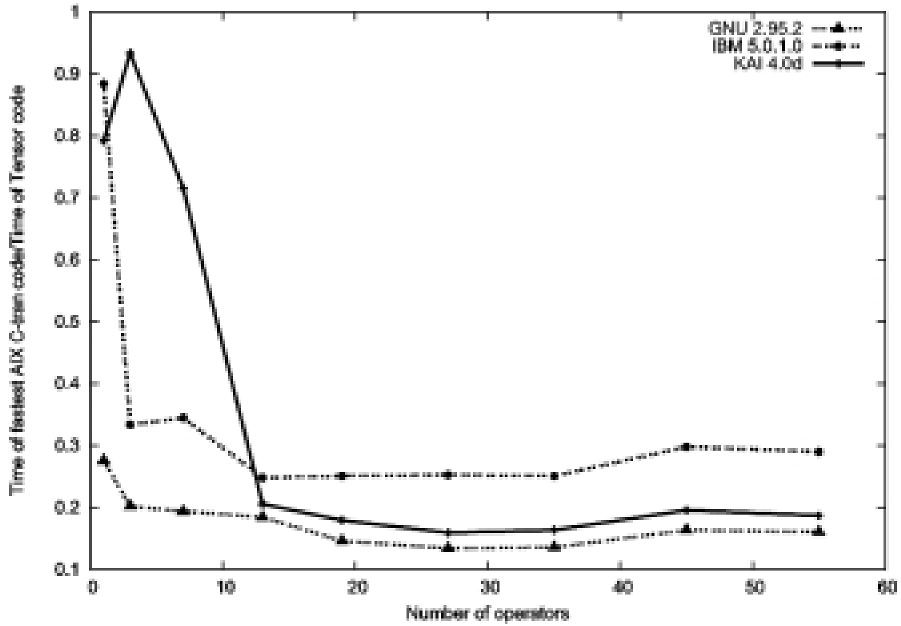


Fig. 4. Relative execution times of fastest C-tran and Tensor1 on AIX for different compiler versions.

```
  T & operator()(int N) {
    return data[N];
  }
  .
  .
  .
};
```

There is a minor wrinkle if we want to use a simple constructor syntax like

```
Tensor1<int,3> T1(1,2,3);
```

If we simply add in the constructors for two and three dimensions, then an unwitting user might write

something like

```
Tensor1<int,3> T1(1,2);
```

and thus not set all of the elements. Worse, they might write

```
Tensor1<int,2> T1(1,2,3);
```

and write off of the end of the array. This could be a source of subtle bugs. Since compile-time checks are better than run-time checks, we solve this with a helper class. For `Tensor1`, this class is

```
template<class T, int Dim> class Ten
sor1_constructor;

template<class T> class Tensor1_constructor
<T,2> {
public:
  Tensor1_constructor(T data[], T d0, T d1) {
    data[0]=d0;
    data[1]=d1;
  }
};
template<class T> class Tensor1_constructor
<T,3> {
public:
  Tensor1_constructor(T data[], T d0, T d1, T
  d2) {
    data[0]=d0;
    data[1]=d1;
    data[2]=d2;
  }
};
// And similarly for 4,5,6,... dimensions.
```

In the `Tensor1` class, we define the constructors to just call `Tensor1_constructor`'s constructor

```
Tensor1(T d0, T d1) {
  Tensor1_constructor<T,Dim>(data,d0,d1);
}
Tensor1(T d0, T d1, T d2) {
  Tensor1_constructor<T,Dim>(data,d0,d1,d2);
}
```

Now, if someone tries to give too many or too few arguments to the constructor, the compiler will not be able to find the correct constructor for `Tensor1_constructor`. The partially specialized versions of `Tensor1_constructor` only have constructors for the correct number of arguments.

Indexing is also much simpler when using arrays, although symmetric and antisymmetric tensors require some attention. A rank 2 symmetric tensor in `Dim` dimensions has (Dim(Dim+1)/2) independent elements, while an antisymmetric tensor has (Dim(Dim-1)/2) independent elements. We store them in a one-dimensional array and translate between the tensor indices and the array index. For `Tensor2_symmetric`, this means that `operator()(int,int)` becomes

```
T& operator()(const int N1, const int N2)
{
  return N1>N2 ? data[N1+(N2*(2*Tensor_Dim
               - N2-1))/2]:
               data[N2+(N1*(2*Tensor_Dim
               - N1-1))/2];
}
```

A similar technique works for antisymmetric tensors

```
T operator()(const int N1, const int N2) const
{
  return N2<N3 ? data[N1][N3-1+(N2*(2*(Tensor
        _Dim12-1)-N2-1))/2] :
        (N2>N3 ? -data[N1][N2-1+(N3*(2*(Tensor
        _Dim12-1)-N3-1))/2] :
         0.0);
}
```

We also modify the `TensorN_Expr` classes so that they carry information about their dimension and type. We can use traits [7] to automatically promote types (e.g. from `int` to `double`, or from `double` to `complex<double>`). We can also make the arithmetic operators dimension agnostic with some template meta-programming [14]. It turns out that the only place where the dimension comes in is in assignment and in implicit summation. For the case of assignment to a `Tensor1_Expr`, the code becomes

```
template<class A, class B, class U, int Dim,
         char i,
         int Current_Dim> inline
  void T1_equals_T1(A &iter,
                    const Tensor1_Expr
                    <B,U,Dim,i>result,
                    const Number<Current_Dim>
                    &N)
{
  iter(Current_Dim-1)=result(Current_Dim-1);
  T1_equals_T1(iter,result,Number
  <Current_Dim-1>());
}
template<class A, class B, class U, int Dim,
char i> inline
void T1_equals_T1(A &iter, const
Tensor1_Expr<B,U,Dim,i> result,
                  const Number<1> &N)
{
  iter(0)=result(0);
}
template<class A, class T, int Tensor_Dim,
int Dim, char i>
template<class B, class U> inline const
Tensor1_Expr<Tensor1<A,Tensor_Dim>,T,Dim,i> &
Tensor1_Expr<Tensor1<A,Tensor_Dim>,T,Dim,i>::
```

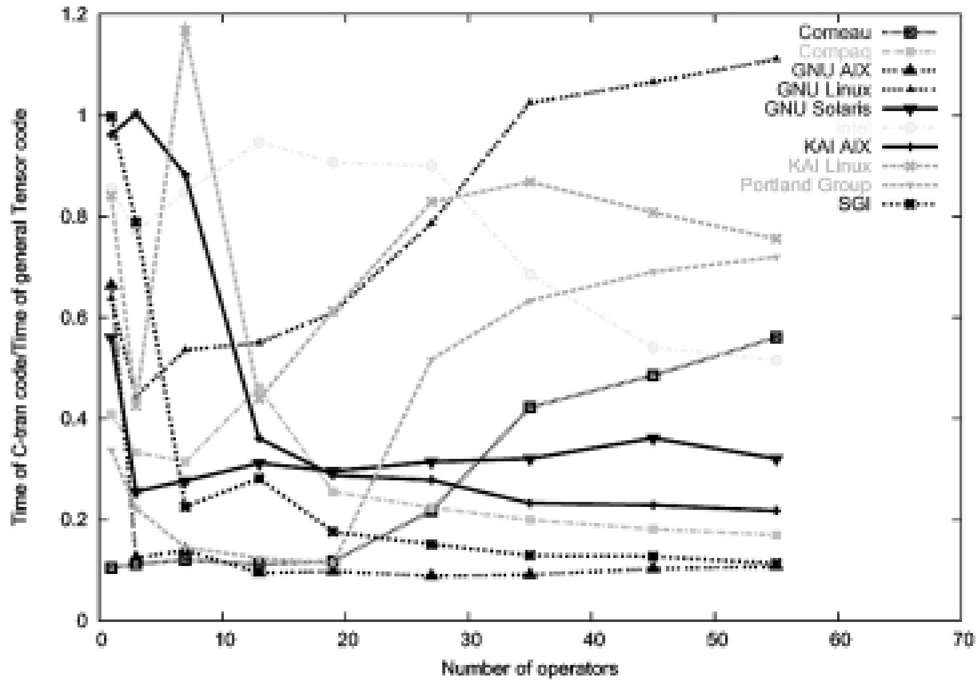


Fig. 5. Relative execution time of C-tran and more general Tensor's.

```
operator=(const Tensor1_Expr<B,U,Dim,i>
&result)
{
  T1_equals_T1(iter,result,Number<Dim>());
  return *this;
}
```

That is, we've used templates to do a simple loop from `Dim-1` to `0`. Defining assignment operators for higher rank tensors as well as the implicit summation functions uses similar loops. Now, if we're trying to follow Buckaroo Banzai across the 8th dimension, we only have to define the constructors for `Tensor1`, `Tensor2`, `Tensor3`, etc. classes for eight dimensions, and all of the `Tensor_Expr` classes and arithmetic operators are ready to use.

In this framework, we can also define `Index` to have a dimension

```
template<char i, int Dim>
class Index{};
```

When creating a `Tensor_Expr`, we can use the dimension of the `Index` rather than the dimension of the `Tensor` to determine `Tensor_Expr`'s dimension. Then if we have a four-dimensional tensor, it becomes simple to manipulate only the lower three dimensional parts by using only three dimensional `Index`'s. There is a danger, though. What if we use a four-dimensional `Index` in a three dimensional `Tensor`? Range-checked builds should catch this kind of error.

We have implemented this generalization [6]. IBM's xlC can not compile it, always aborting with an Internal Compiler Error. Also, KCC can't fully optimize complicated expressions in the first benchmark as it could with the simpler version of the library, leading to code that runs hundreds of times slower. Interestingly enough, the TinyVector classes in Blitz [11] are also templated on type and dimension, and complicated expressions can not be fully optimized in that kind of benchmark as well.

However, the performance in the second benchmark is not affected in the same way. Figure 5 shows the relative execution times for C-tran versus the more general Tensor's, and Fig. 6 directly compares the two versions of the Tensor library.

The results are generally mixed, although the KAI, Portland Group, and Intel compilers generally do better while the Comeau compiler does worse. Once again, the non-x86 compilers do not perform very well. However, the overall conclusions from the last section are unchanged. This is nice, in the sense that using a more general library doesn't necessarily cause another hit in performance.

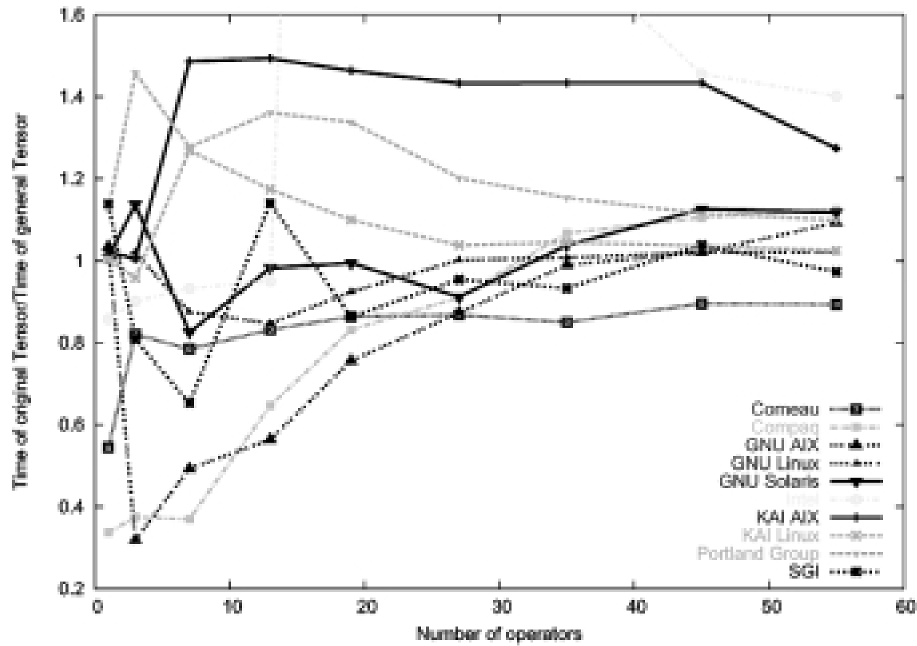


Fig. 6. Relative execution time of simple and more general Tensor's.

6. Conclusion

We have described the design of a freely available, high performance tensor library [6]. It uses a number of modern C++ template tricks, supported by most, but not all, compilers, to produce generic, flexible, and fast code. These tricks include expression templates [13], template metaprograms [14], and traits [7], mixed in with a number of helper classes. They allow the library to express indices, tensors, tensor expressions, binary operators on these expressions, internal and external contractions, reduced rank tensors, and tensor symmetries in an efficient, type-safe framework, generalized for any dimension or type, using natural notation.

However, the original promise of expression templates as a way to get away from C-tran is not completely fulfilled. Although the syntax is much improved, there are still cases where a programmer must resort to at least some manual loops in order to get maximum performance. Even with this work, there are still performance penalties, sometimes severe, which vary from problem to problem, although in some cases making a library more general and using template techniques in more places can improve performance. In particular, non-x86 compilers seem to do a bad job of optimizing complicated template expressions. This is probably just a reflection of the relative efforts that has been put into optimizing for the x86 platform.

Despite these caveats, for our General Relativity application expression templates were a huge win. Compared to simple tensor classes, template techniques enabled faster, smaller executables, though at the cost of longer compilation times and stringent compiler requirements. Even compared to C-tran, it is not clear that there is a significant speed penalty. What is clear is that the program would never have been finished without the syntactic simplicity afforded by tensor classes of some sort. Furthermore, compiler requirements have become much less onerous as C++ compiler technology has caught up with the ISO standard.

Acknowledgements

We gratefully acknowledge the help of Comeau computing in providing a copy of their compiler for evaluation. This work was supported in part by NSF grant PHY 97-34871 and DOE grant DE-FC02-01-ER41186. An allocation of computer time from the Center for High Performance Computing at the University of Utah is gratefully acknowledged. CHPC's IBM SP system is funded in part by NSF Grant #CDA9601580 and IBM's SUR grant to the University of Utah. This research was supported in part by NSF cooperative agreement ACI-9619020 through computing resources provided by the National Partnership for Advanced Computational Infrastructure at the San Diego Supercomputer Center.

Appendix: Compiler options

Comeau	–Drestrict= –O3 –remove_unneeded_entities –pending_instantiations=100
Compaq	–std ansi –model ansi –nousing_std –noexceptions –nortti –Drestrict=__restrict –assume noptrs_to_globals –assume whole_program –assume noaccuracy_sensitive –inline all –fast –O5 –non_shared –tune host –pending_instantiations 1000 –nocleanup
GNU	–O3 –ffast-math –finline-functions -finline-limit-1000 –funroll-loops –ftemplate-depth-100 –Drestrict=
IBM	–Drestrict= –O3
Intel	–restrict –O3 –xi –ipo –Qoption,c,-ip_ninl_max_stats=10000
KAI 3.4/ AIX	+K3 –restrict –no_exceptions –inline_auto_space_time=750000000000000 –inline_implicit_space_time=200000000000000 –inline_generated_space_time=40000000000000.0 –inline_auto_space_time=100000000000000.0 –max_pending_instantiations 100 –qmaxmem=100000
KAI 4.0d/ Linux	+K3 –restrict –no_exceptions –inline_auto_space_time=750000000000000 –inline_implicit_space_time=200000000000000 –inline_generated_space_time=40000000000000.0 –inline_auto_space_time=100000000000000.0 –max_pending_instantiations 100
Portland Group	–Drestrict= –fast –Minline=levels:10 –no_exceptions
SGI	–LANG:std –LANG:restrict=ON –64 –O3 –LANG:exceptions=OFF –IPA:space=1000000000 –IPA:plimit=1000000000 –OPT:unroll_times_max=100000 –OPT:unroll_size=1000000 –INLINE=all –IPA:alias=ON

References

[1] W. Bangerth, G. Kanschat and R. Hartmann, *Deal.II*, http://gaia.iwr.uni-heidelberg.de/deal/.

[2] N. Gaspar, http://www.openheaven.com/believers/neil_gaspar/t_class.html.

[3] B. Jeremic, *nDarray*, http://civil.colorado.edu/nDarray/.

[4] W. Landry, http://superbeast.ucsd.edu/landry/FTensor.tar.gz.

[5] W. Landry and S. Teukolsky, http://xxx.lanl.gov/abs/gr-qc/9912004.

[6] W. Landry, http://www.oonumerics.org/FTensor/.

[7] N. C. Myers, Traits: a new and useful template technique, *C++ Report* **7**(5) (1995).

[8] TensorSoft, *GRPP*, http://home.earthlink.net/tensorsoft/.

[9] *POOMA*, http://www.acl.lanl.gov/pooma/.

[10] R. Tisdale, *SVMT*, http://www.netwood.net/edwin/svmt/.

[11] T. Veldhuizen, *Blitz*, http://www.oonumerics.org/blitz.

[12] T. Veldhuizen, http://oonumerics.org/blitz/benchmarks/Origin-2000-SGI/.

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

[14] T. Veldhuizen, Using C++ template metaprograms, *C++ Report* **7**(4) (1995), 36–43.

[15] Comeau Computing, http://www.comeaucomputing.com.

[16] HP/Compaq, http://www.tru64unix.compaq.com/cplus/.

[17] GNU gcc, http://gcc.gnu.org.

[18] Kuck and Associates, Inc. (KAI), http://www.kai.com, Note that KAI has been acquired by Intel, and it is no longer possible to buy KAI C++.

[19] IBM, http://www.ibm.com/software/ad/vacpp/, xlC is the batch part of Visual Age C++.

[20] SGI,http://www.sgi.com/developers/devtools/languages/C++.html.

[21] Intel, http://www.intel.com/software/products/compilers/.

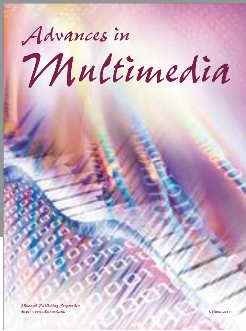
[22] Portland Group, http://www.pgroup.com.

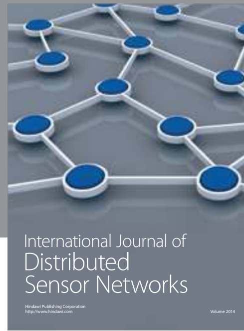
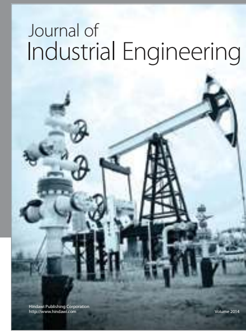
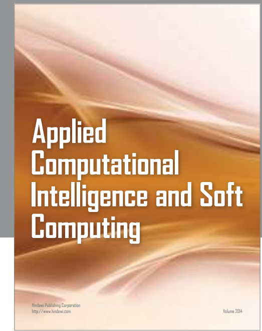
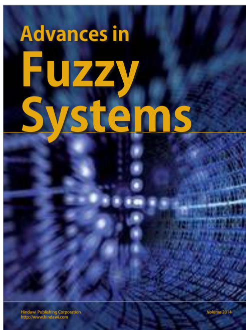
[23] Sun,http://wwws.sun.com/software/sundev/suncc/index.html.

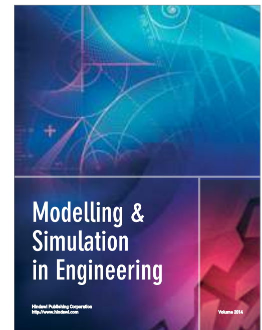
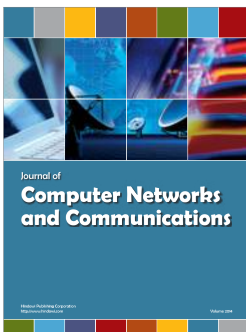
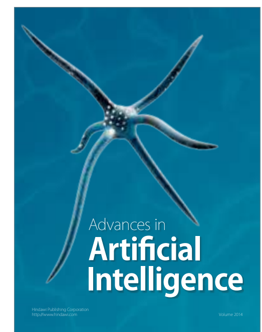
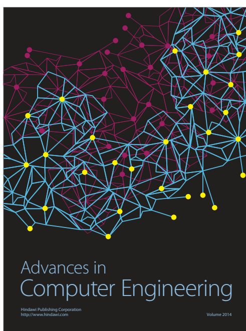
[24] Microsoft, http://msdn.microsoft.com/library/en-us/vclang/html/vclrf1454ClassTemplatePartialSpecialization.asp.

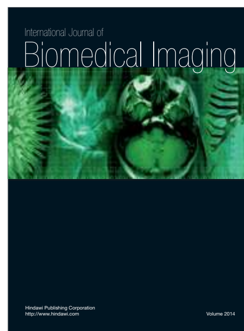
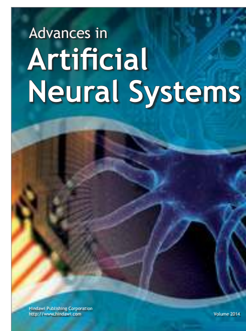
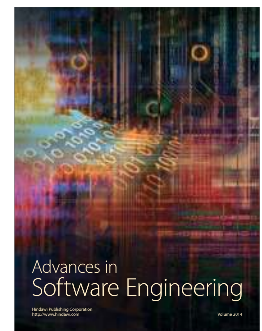
[25] GNU gcc, http://gcc.gnu.org/gcc- 3.1/criteria.html.

[26] B. Stroustrup, *The C++ Programming Language*, 3rd edition, Addison Wesley, section 22.4.7, 1997, pp. 675.

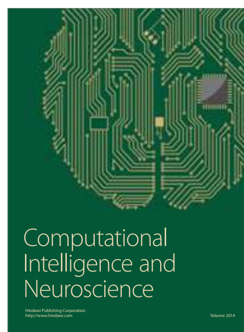

Advances in
Multimedia

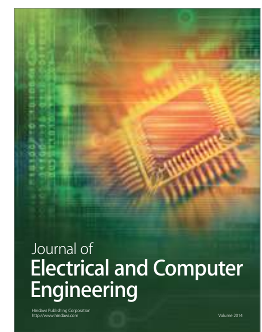
The Scientific
World Journal

International Journal of
Distributed
Sensor Networks

Journal of
Industrial Engineering

Applied
Computational
Intelligence and Soft
Computing

Advances in
Fuzzy
Systems

Hindawi
Submit your manuscripts at
http://www.hindawi.com

Modelling &
Simulation
in Engineering

Journal of
Computer Networks
and Communications

Advances in
Artificial
Intelligence

Advances in
Computer Engineering

International Journal of
Computer Games
Technology

International Journal of
Biomedical Imaging

Advances in
Artificial
Neural Systems

Advances in
Software Engineering

Journal of
Robotics

Advances in
Human-Computer
Interaction

Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering