

Implementing a High Performance Tensor Library

Walter Landry
University of Utah
landry@physics.utah.edu

September 1, 2001

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 either usability or 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

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

Alternatively, we can write it as

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

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

$$R_{ijkl}^i = dG_{jkl}^i - dG_{lkj}^i + G_{jk}^m G_{ml}^i - G_{lk}^m G_{mj}^i.$$

There are some subtle differences between tensors with indices that are upstairs (like T^i), and tensors with 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 computers are supposed to do for you. 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][2][3][4][5][6]). With one exception, they are all either difficult to use (primarily, not providing implicit summation), or they are not efficient. GRPP [6] solves this conundrum with a proprietary mini-language, making it difficult to customize and extend. With expression templates, it is possible to create a library within the C++ language which is both efficient and relatively easy to use.

2 Implementations

2.1 The Easy-to-Implement, Inefficient Solution with Nice Notation

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 ($temp1 = D_j E_j$, $temp2_i = C_i * temp1$, $temp3_i = B_i + temp2_i$) requiring $7N$ 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 The Hard-to-Implement, Somewhat Inefficient Solution with Nice Notation

This is the sort of problem for which template methods are well-suited. Using expression templates [8], we can write

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

and have the compiler transform it into a 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. This code can run four or more times faster 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 multiple resolutions of the same grid (as happens with adaptive grid methods).

As an aside, the Blitz library [1] uses this approach. On the benchmark page for the Origin 2000/SGI C++ [7], 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 slowdowns. It even mentions explicitly (after loop test #14) "The lack of loop fusion really hurts the C++ versions."

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

2.3 The Hard-to-Implement, Efficient Solution with only Moderately Nice Notation

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 did 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 pointer class is

```
class Tensor1_ptr
```

```

{
    mutable double *x, *y, *z;
public:
    void operator++()
    {
        ++x;
        ++y;
        ++z;
    }
    \\ Indexing, assignment, initialization 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_ptr`'s. The indexing operators for `const Tensor1_ptr` returns a `double`, not `double *` or `double &`, so the actual data can't be changed. This keeps the data logically `const`, while allowing us to look at all of the points on the grid for that `const Tensor1_ptr`.

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;
}

```

An example which mixes the pointer and non-pointer tensor classes is

```

void f(const Tensor2_ptr T, const Tensor1_ptr P,
      Tensor1_ptr Q, const int N)
{
    for(int n=0;n<N;++n)
    {

```

```

    Tensor2 T_symmetric;
    T_symmetric(i,j)=(T(i,j)+T(j,i))/2;
    Q(i)=T_symmetric(i,j)*P(j);
    ++Q;
    ++T;
    ++P;
}
}

```

This function symmetrizes the matrix `T` and multiplies it by `P`, putting the result in `Q`. The body inside the loop can also be simplified to remove the named temporary `T_symmetric`

```

    Q(i)=(T(i,j)+T(j,i))*P(j)/2;
    ++Q;
    ++T;
    ++P;

```

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 attempt to iterate through a grid twice. That is why this function `f()` passes the tensors by value, and not by reference. Otherwise, the pointers in `Q`, `T`, and `P` would have been iterated to the end. The parent function calling `f()` would then not be able to use its copies of `Q`, `T`, and `P`.

In practice, these were not serious problems, because most of the logic of the program is in the manipulation of local named variables. Only a few variables (the input and output) need to be explicitly iterated. Even those that were iterated were locally constructed from global arrays given by another library that managed multiprocessor computations.

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 How Well Does it Work?

We have implemented the first (inefficient) method and the third (efficient) method [10]. We did not attempt to implement the second method, because it was clear that it could not be as efficient as the second method, while still being a difficult chore to implement. We have also 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.

For our General Relativity code, the efficient method is about two times faster than and uses about a third the memory of the inefficient method. However, not all compilers support

GNU gcc 2.95.2/Linux x86 GNU gcc 2.95.3/Solaris Sparc	Doesn't Fully Optimize
GNU gcc 2.95.2/Alpha	Can't handle long mangled names with non-GNU assembler
Portland Group pgCC 3.2/Linux x86	Can't handle long mangled names no <cmath>
KAI KCC 4.0d/Linux x86 KAI KCC 4.0d/AIX	Optimizes as well as anything
IBM xlc 5.0.1.0/AIX	Doesn't fully optimize, ICE's
Sun CC 6.1/Solaris Sparc	Doesn't support non-type template parameters
SGI CC 7.3.1.1m/Irix	Doesn't fully optimize, no <cmath>

Table 1: Compiler Comparison

enough of the standard to compile the efficient library, while the inefficient method works with almost any compiler. A comparison of 9 combinations of compiler and operating system is shown in Table 1.

Some of the assemblers had trouble with the long mangled names produced by the compiler. Surprisingly, not all of the compilers made <cmath> available, although it is easy to work around that. xlc 5.0.1.0 seems to be immature, with a remarkable number of cases of Internal Compiler Errors (ICE's). The Sun compiler seems to be useless for these kinds of template techniques. 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 in order to do it. Depending on how complicated the expressions is, not fully optimizing the expressions can slow down the code in that expression by factors of 10 or more. However, that didn't seem to matter that much for the General Relativity code. Although the expressions were extremely complicated, both gcc and xlc ran only 10-20% slower than KCC. This suggests that the code is still dominated by the cost of moving data between the memory and the processor. Your mileage may vary.

4 Extending the Library

An overly clever reader may have looked at the rough declaration of `Tensor1_ptr` 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>`. 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 then specialize for each dimension and whether the type is a pointer

```
template<class T, int Dim> class Tensor1;
template<class T> class Tensor1<T,2> {
```

```

    T x, y;
    .
    .
    .
}
template<class T> class Tensor1<T*,2> {
    mutable T *x, *y;
    .
    .
    .
}
template<class T> class Tensor1<T,3> {
    T x, y, z;
    .
    .
    .
}

```

and so on. We can even make the arithmetic operators dimension agnostic with some template meta-programming [9]. Then, if you're trying to follow Buckaroo Banzai across the 8th dimension, you only have to define the Tensor1, Tensor2, Tensor3, etc. classes for eight dimensions, and all of the arithmetic operators are ready to use.

We have implemented this generalization [11], but it complicates the logic enough such that even KCC can't fully optimize complicated expressions. It also makes use of some constructs which are not supported by gcc. Interestingly enough, the TinyVector classes in Blitz [1] are also templated on type and dimension, and complicated expressions can not be fully optimized as well.

5 Conclusion

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. Finally, it is also not always a clear win to make code as generic as possible, as that can make the job of the compiler impossibly hard.

Acknowledgements

This work was supported in part by NSF grant PHY 97-34871.

References

- [1] Todd Veldhuizen, Blitz, <http://www.oonumerics.org/blitz>
- [2] Neil Gaspar, http://www.openheaven.com/believers/neil_gaspar/t_class.html
- [3] Boris Jeremic, nDarray, <http://civil.colorado.edu/nDarray/>
- [4] Wolfgang Bangerth, Guido Kanschat, Ralf Hartmann, Deal.II, <http://gaia.iwr.uni-heidelberg.de/~deal/>
- [5] Robert Tisdale, SVMT, <http://www.netwood.net/~edwin/svmt/>
- [6] TensorSoft, GRPP, <http://home.earthlink.net/~tensorsoft/>
- [7] <http://oonumerics.org/blitz/benchmarks/Origin-2000-SGI/>
- [8] Todd Veldhuizen, "Expression Templates," *C++ Report*, Vol. 7 No. 5 (June 1995), pp. 26-31
- [9] Todd Veldhuizen, "Using C++ template metaprograms," *C++ Report*, Vol. 7 No. 4 (May 1995), pp. 36-43.
- [10] <http://www.physics.utah.edu/~landry/FTensor.tar.gz>
- [11] http://www.physics.utah.edu/~landry/FTensor_new.tar.gz