Implementing a High Performance Tensor Library

Walter Landry

University of Utah

University of California, San Diego

Terascale Supernova Initiative

wlandry@ucsd.edu

Overview

- 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.

- In this talk, I'm going to describe the design of a friendly, high performance tensor library.

- We find that template methods allow us to create a powerful, flexible library, including features not found in other libraries, but performance did suffer.

Introduction to Tensors

- 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 multiplying a vector P by a matrix T, yielding a vector Q

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

  Writing the equations out explicitly

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

- Alternately, 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.

Summation Notation

- Einstein introduced the convention that if an index appears in two tensors that multiply each other, then that index is implicitly summed.

- Using this Einstein summation notation, the matrix-vector multiplication becomes simply

  $$Q_i = T_{ij} P_j$$

  This mostly removes the need to write the summation symbol $\sum_{j=x,y,z}$. This implicit summation is also called contraction.

- 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

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

Expressing this in Code

- Now consider expressing this equation in code. We could use multidimensional arrays and start writing lots of loops

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

- 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 set all of the elements and do all of the contractions for you.

Implementation

- To illustrate our basic design, we start with rank 1 tensors. 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(1138)` will return the same result as `A(2)`. We could introduce a checked version using `#ifdef DEBUG` macros, but it has not been done yet.

- 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, because they transparently provide reasonably 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 it's index is.

- 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);
    }
};
```

- To create a `Tensor1_Expr`, we overload the function operator 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);
```

  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 want the syntax to be `A(i)=B(i)`. This implies that we are actually assigning one `Tensor1_Expr` to another.

- So we have to specialize `Tensor1_Expr` to the case where 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=()`.

- This provides strong compile-time checking of tensor expressions.

- Generalizing this to higher rank tensors is 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 specialize it for `TensorN`'s and define an equal's operator.

Arithmetic Operators

- Now we want to do something really useful. We want to add two `Tensor1`'s together. This is where expression templates really comes into play.

- We do this by creating a helper class `Tensor1_plus_Tensor1`.

- 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 `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<Tensor1,'i'>`) it contains.

- The `Tensor1_Expr`'s pass the call onto the `Tensor1`, and we 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(j)` creates a new `Tensor2` with indices of `i` and `j`.

- To do this, 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);
  }
};
```

- 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));
}
```

Implicit Summation

- The preceding work is not really that interesting. Blitz 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.

External Contraction

- External contraction is when the index of one tensor contracts with the index of another tensor. Consider the simple contraction of two rank 1 tensors

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

We want this to be equivalent to

```
Tensor1 A,B;
double result=A(0)*B(0)+A(1)*B(1)+A(2)*B(2);
```

- 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));
}
```

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);
```

  We want this to be 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> index1,
                  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)`.

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)`(Symmetric), or `A(i,j)=-A(j,i)`(Antisymmetric).

- It can be quite advantageous to take advantage of these symmetries because it reduces 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.

Symmetric Tensors

- It turns out that implementing a symmetric tensor is quite simple.
- First, we define a class (`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 symmetry 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`.

- We simplify the equals operator so that it only sets the elements we have.

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

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)`).

- If we just want to read it, that is no problem. We just return zero.

- However, what if we want to write to it (such as in `A(0,0)=1`)?

- Introducing a run time error will make it harder for the compiler to optimize expressions.

- The kludge that we have come up with is to return a dummy variable named `zero`.
  - This variable is a member of the `TensorN_Antisymmetric` class, and is initialized to 0.
  - Because a read will often look like a write, we will still get the right answer in those cases.
  - However, if `zero` is actually written to, it will ruin this trick.
  - It therefore places a burden on the application programmer not to assign to identically zero elements.

Reduced Rank Tensors

- Expressions like `A(i)=T(0,i)` can sometimes pop up.

- To support this, we make another helper class.

```
template<class A, char i>
class Tensor2_number_0
{
  const A &iterA;
  const int N;
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, const int NN): iterA(a), N(NN) {}
};
```

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

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

- The end result of all of this is that when we write statements like

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

we create a `Tensor1_Expr<Tensor2_number_0<Tensor2,'i'>,'i'>` which then gets assigned to the `Tensor1_Expr<Tensor1,'i'>` created by `A(i)`.

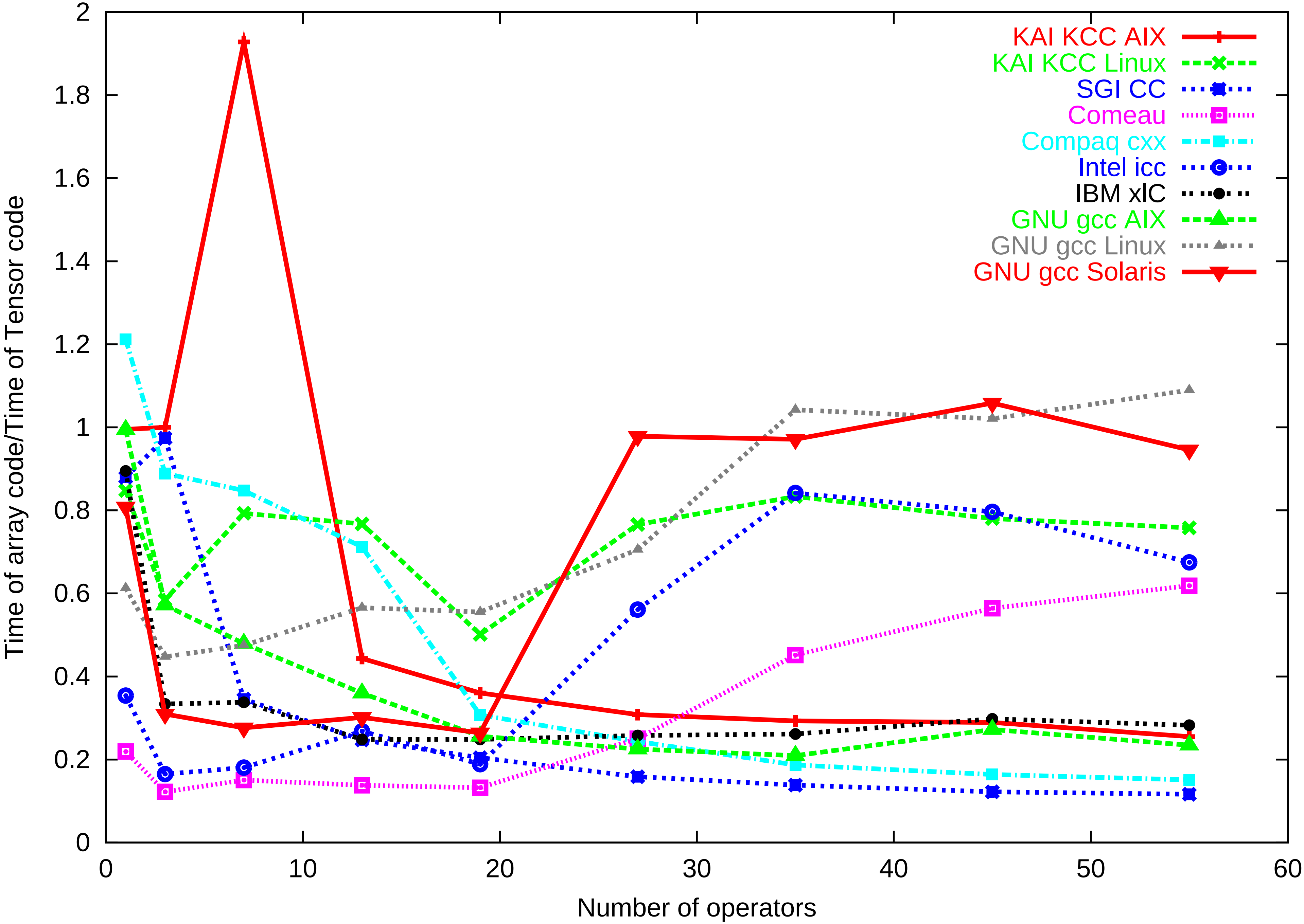
Compiler/Operating System	Compiles the library?
Comeau como 4.2.45.2 + libcomobeta14/Linux x86 Compaq cxx 6.3/Tru64 GNU gcc 2.95.2/Linux x86, 2.95.3/Solaris, 2.95.2/AIX KAI KCC 4.0d/Linux x86, 4.0d/AIX	Yes
IBM xlC 5.0.1.0/AIX	Yes with occasional ICE's
SGI CC 7.3.1.1m/Irix	Somewhat-no <cmath> and can't override template instantiation limit
Intel icc 5.0/Linux x86	Somewhat-uses excessive resources and can't override template instantiation limit
Portland Group pgCC 3.2/Linux x86	No, can't handle long mangled names, no <cmath>
Sun CC 6.1/Solaris Sparc	No, doesn't support partial specialization with non-type template parameters

Performance

- We've written some simple benchmarks. One shows that, with the KAI compiler, this library is just as fast as using simple arrays. The other compilers can be hundreds of times slower.

- This is good evidence that we didn't make any serious performance errors in our implementation.

- A more complicated, but still simple, benchmark shows very different results



KAI KCC AIX
KAI KCC Linux
SGI CC
Comeau
Compaq cxx
Intel icc
IBM xlC
GNU gcc AIX
GNU gcc Linux
GNU gcc Solaris
Time of array code/Time of Tensor code
Number of operators
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0
10
20
30
40
50
60

Extending the Library

- A reader with foresight may have looked at the declaration of `Tensor1` and thought that hard coding it to be made up of `double`'s 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

```
template<class T, int Dim> class Tensor1;
template<class T> class Tensor1<T,2> {
  T x, y;
  .
  .
  .
}
template<class T> class Tensor1<T,3> {
  T x, y, z;
  .
  .
  .
}
```

- Alternately, we could use arrays, and then we wouldn't have to specialize for each dimension. However, that doesn't work very well for tensors with symmetries.

- We use traits to automatically promote types (e.g. from `int` to `double`, or from `double` to `complex<double>`).

- We can even make the arithmetic operators dimension agnostic with some template meta-programming.

- 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.

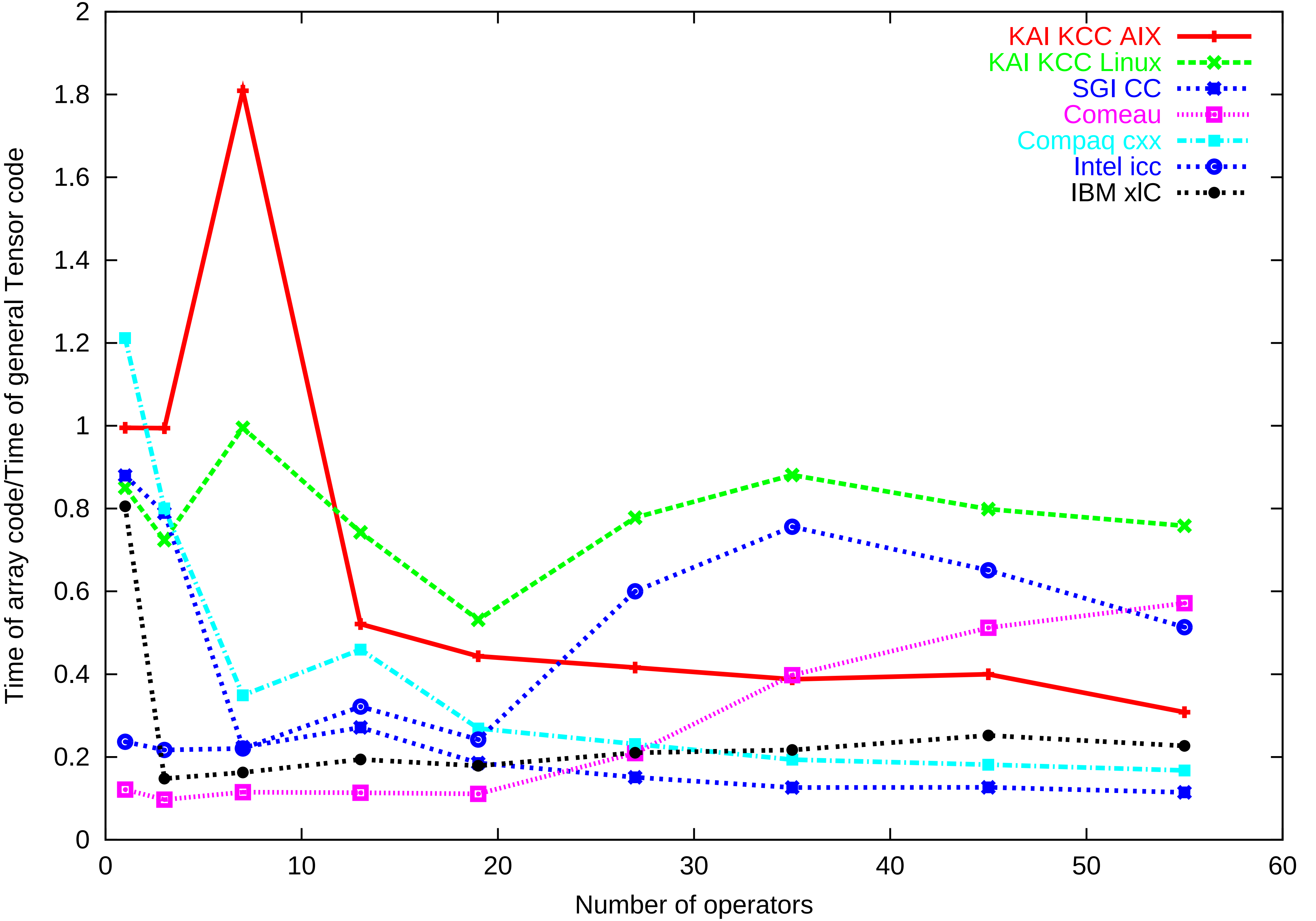
- We can also define `Index` to have a dimension

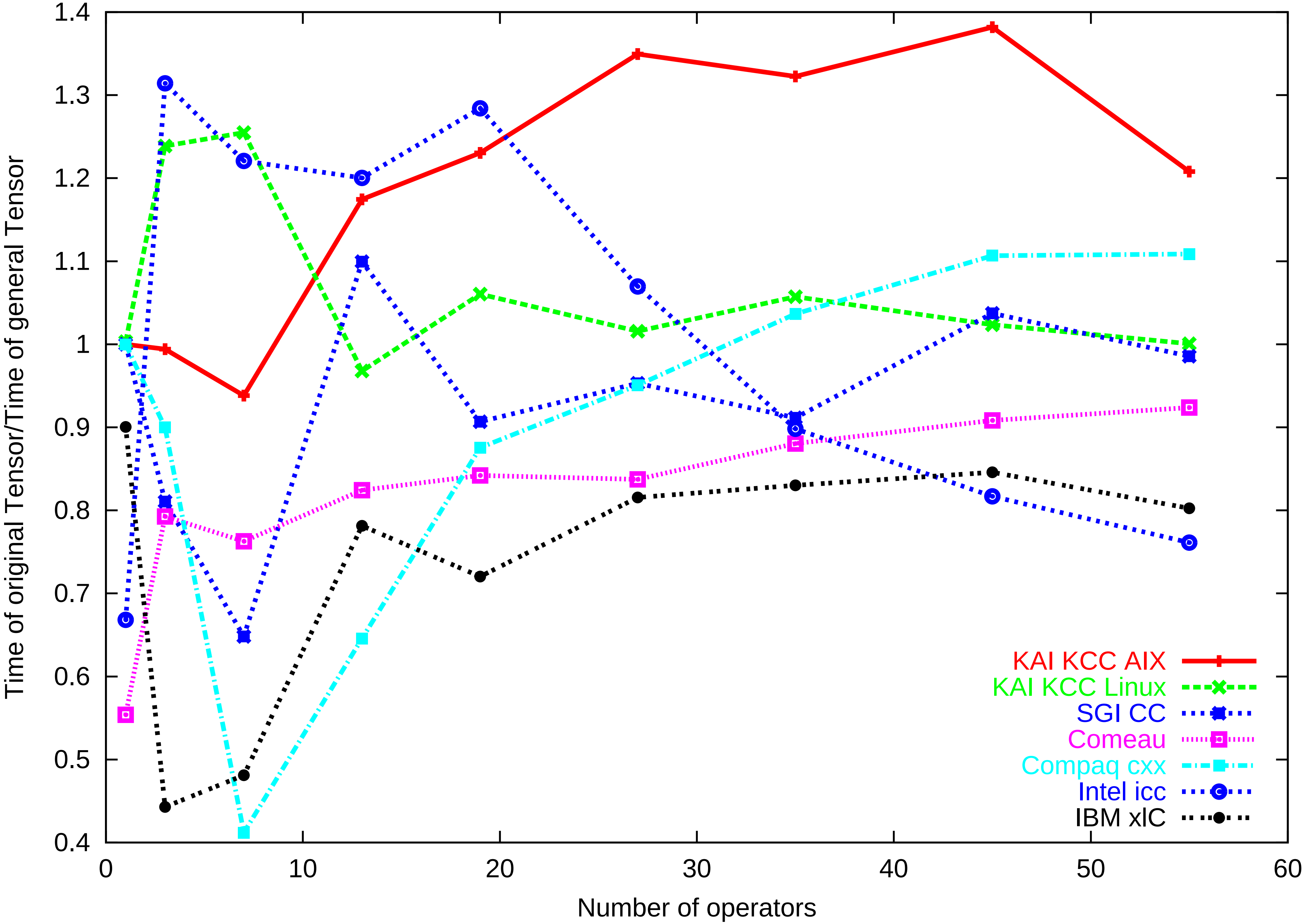
```
template<char i, int Dim>
class Index{};
```

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`? Without range checking, this kind of bug can go undetected for a long time.

- We have implemented this generalization. It uncovers a deficiency in the template support by gcc, so it can't compile it. The performance is similar.


KAI KCC AIX
KAI KCC Linux
SGI CC
Comeau
Compaq cxx
Intel icc
IBM xlC
Time of array code/Time of general Tensor code
Number of operators
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
0
10
20
30
40
50
60



Time of original Tensor/Time of general Tensor
Number of operators
KAI KCC AIX
KAI KCC Linux
SGI CC
Comeau
Compaq cxx
Intel icc
IBM xlC
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4
0
10
20
30
40
50
60

Conclusion

- We have described a high performance tensor library that supports arithmetic operators, implicit summation, and reduced rank with tensors of arbitrary dimension and type, and with any kind of symmetry or antisymmetry.

- Compilers have gotten much better at compiling these libraries. Some of the compilers that were hopeless with these methods a year ago now do reasonably good jobs.

- The library provides reasonable performance, although simple arrays will almost always be faster, sometimes significantly.

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. 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.