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 examine 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.

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

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

- 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.

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

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

Evaluation on a Grid

- Now consider evaluating this equation on an array with N points. 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.

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. Consider evaluating the expression

$$A_i = B_i + C_i \left(D_j E_j \right)$$

```
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.

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, 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]);
    }
```

- 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.

The Downside

- This will have both nice notation and efficiency *for this expression*. What about a group of expressions?

- Consider inverting a symmetric, 3x3 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;
```

```
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.
    .
    .
    .
  }
```

- `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`.

- For various reasons, it will probably never be possible for compilers to fuse all of the loops and remove extraneous temporaries themselves.

- The POOMA library uses an approach which should solve some of these problems. It defers calculations and then evaluates them together in a block. However, it still requires storage for named temporaries.

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

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 proceed 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.
  }
```

- The matrix inversion example then becomes

```
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 contracts that with `P`, putting the result in `Q`.

- This solution is not ideal and has a few hidden traps, but is certainly better than C-tran.

- 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 equations with indices. But for generic arrays, there are no indices. A solution like this would look almost identical to C-tran.

How Well Does it Work?

- We found that, when compiled with KAI's KCC compiler on an IBM SP2, the efficient library runs about twice as fast and uses a third of the memory of the inefficient library. When compiled with GNU gcc or IBM's xlC, the efficient library code was 10-20% slower than when compiled with KCC.

- However, not all compilers support enough of the standard to compile the efficient library, while the inefficient method works with almost any compiler.

Compiler/Operating System	Compiles efficient 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

- We made two small benchmarks to test how well the compilers could optimize away the overhead of the expression templates. The first is a simple loop

```
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 that used normal arrays instead of the Tensor1 class, 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 in order to do it.

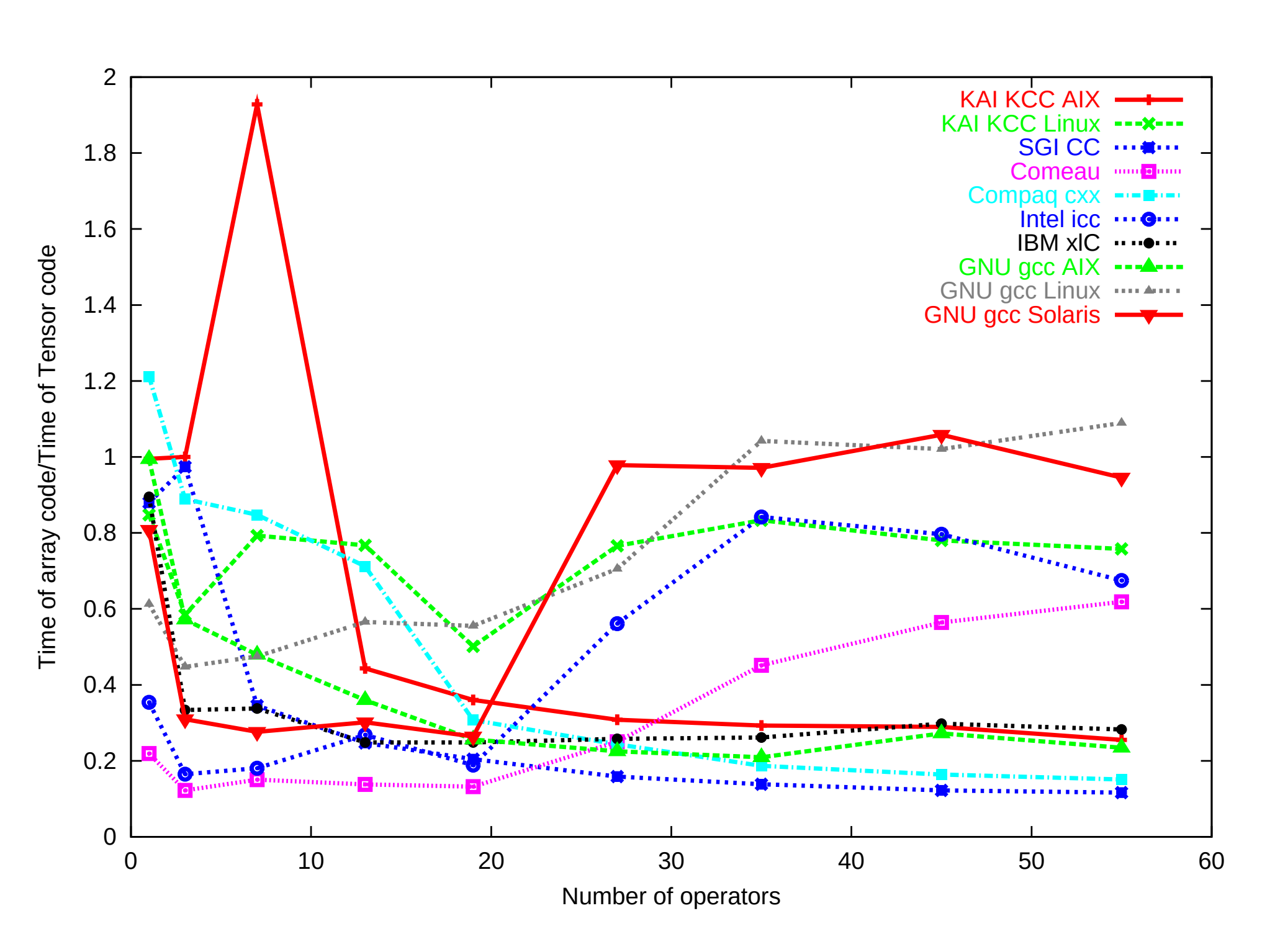
- This is good evidence that we didn't make any serious optimization errors in implementation.

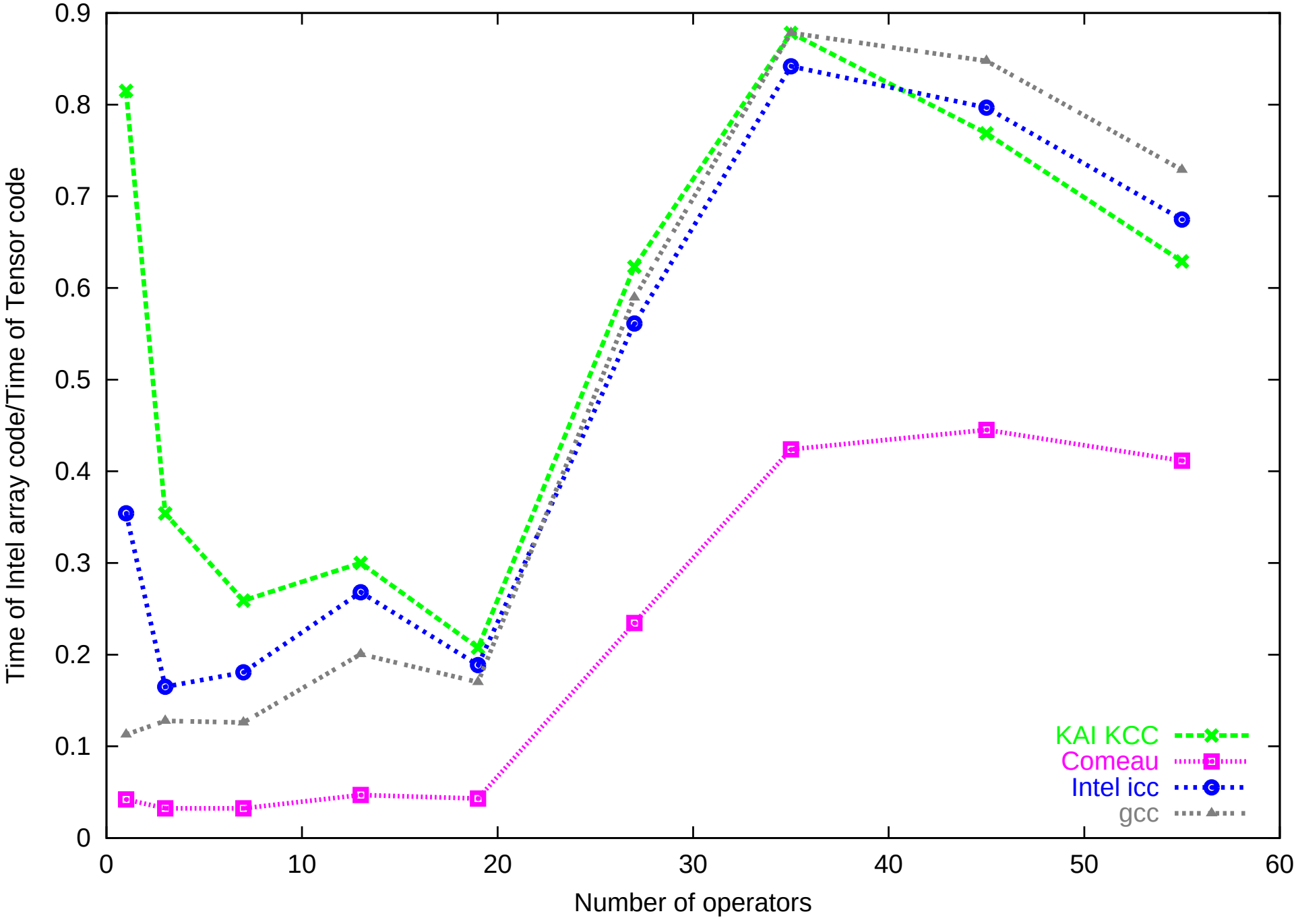
- For the other compilers, the slowdown increased with the number of expressions, becoming more than 100 times slower than the version using arrays.

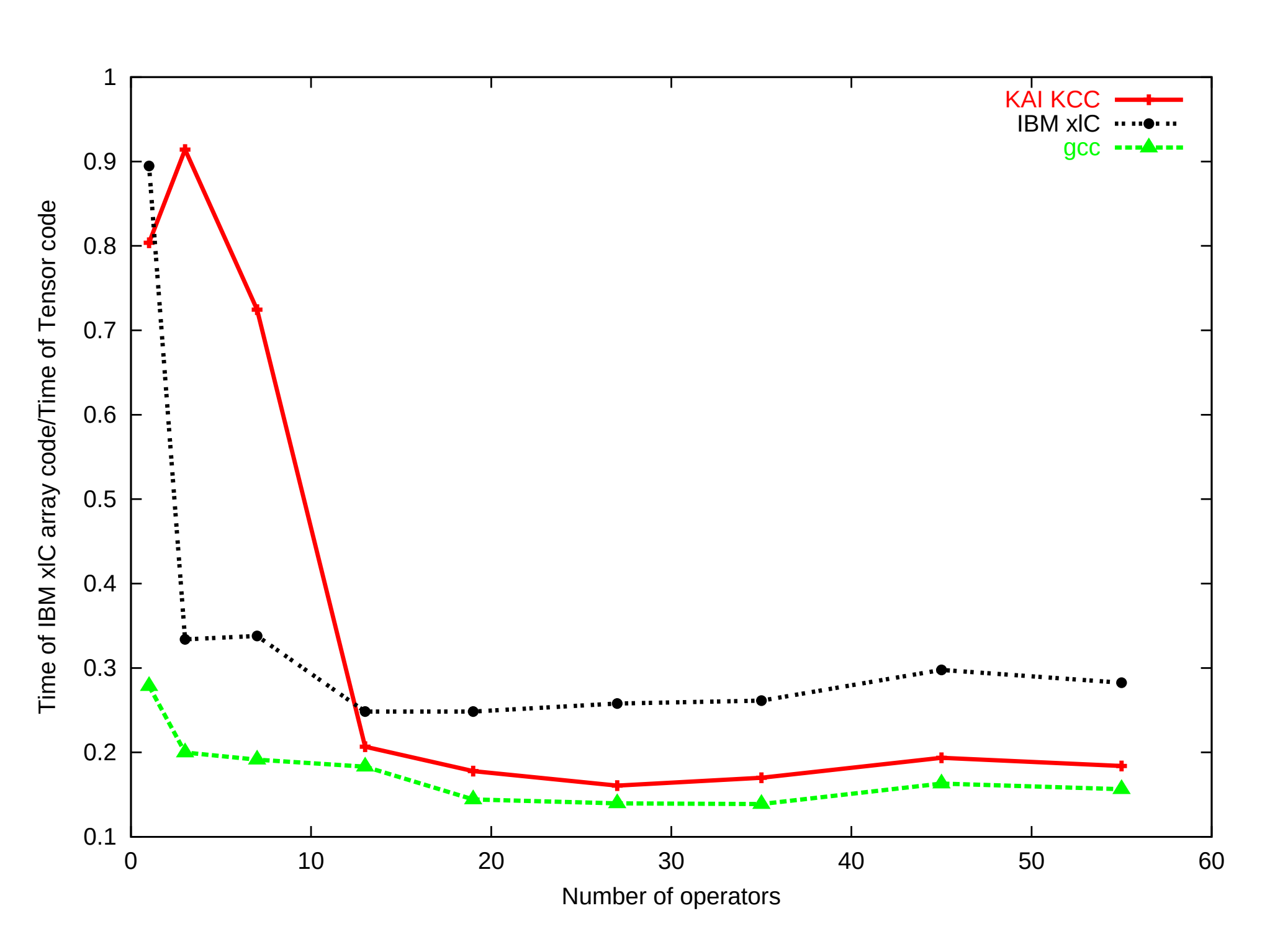
- This benchmark is rather simplistic, so we created a second benchmark.

```
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;
  }
```


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


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
10
20
30
40
50
60
Time of Intel array code/Time of Tensor code
Number of operators
KAI KCC
Comeau
Intel icc
gcc


KAI KCC
IBM xlC
gcc
Time of IBM xlC array code/Time of Tensor code
Number of operators
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
10
20
30
40
50
60

- The first thing to notice in these plots is that none of the compilers can optimize the expressions completely. Even for the best compilers, the Tensor class can run much slower than ordinary arrays.

- However, it is unclear how much this matters in real codes. Writing the equivalent of our General Relativity code using ordinary arrays would be extremely tiresome and error-prone.

- However, as noted earlier, the differences between KCC, xlC, and gcc for our General Relativity code were only 10-20%. Unfortunately, this might just be because they are all terrible at optimizing. Your mileage may vary.

Extending the Library

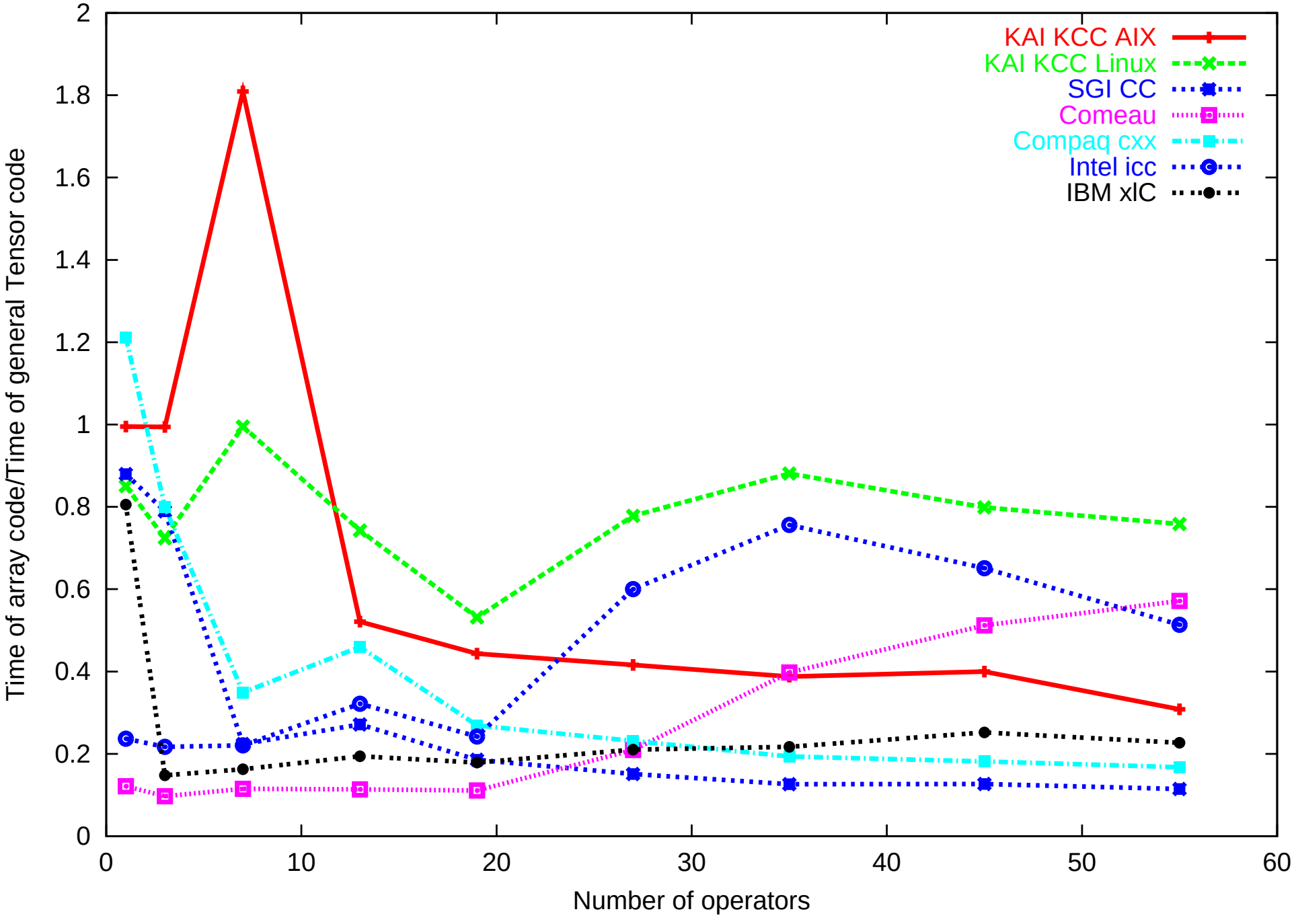
- A reader with foresight may have looked at the rough declaration of `Tensor1_ptr` and thought that hard coding it to be made up of `double*` is rather short sighted. It is not so difficult to envision the need for tensors made up of `int`'s or `complex<double>`.

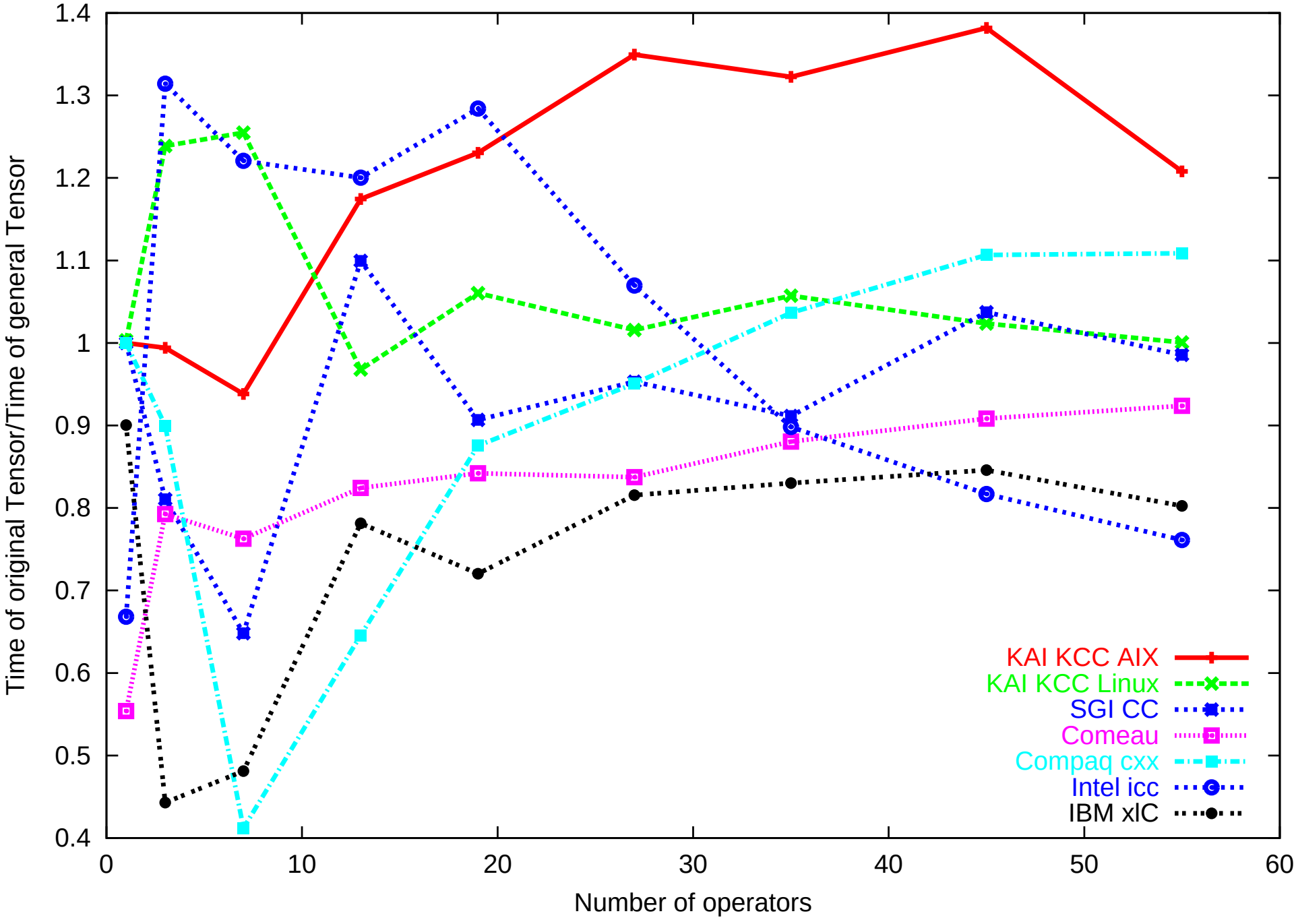
- It might also be nice to use two or four dimensional tensors (so a Tensor1 would have 2 or 4 elements, a Tensor2 would have 4 or 16 elements).

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

- 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 are also templated on type and dimension, and complicated expressions can not be fully optimized in that kind of benchmark as well.


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
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50
60
KAI KCC AIX
KAI KCC Linux
SGI CC
Comeau
Compaq cxx
Intel icc
IBM xlC

- The results are generally mixed, although KCC seems to do better while como and xlC do worse.

- However, the overall conclusions from the last section are unchanged.

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.

- Even with this work, there are still performance penalties which vary from problem to problem.

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.