Editors: Paul F. Dubois, paul@pfdubois.com
David Beazley, beazley@cs.uchicago.edu

A New Look at Expression Templates for Matrix Computation

By Robert C. Kirby

C++ offers programmers the ability to redefine mathematical operations for their own types, such as matrices. Although older approaches to this created many needless temporary objects, a new approach—expression templates—has emerged. This approach bypasses many of the temporary objects and leads to elegant code that's competitive with traditional implementations in C or Fortran. Such mathematical syntax is particularly attractive in matrix computations, as indicated by the success of Matlab and its freeware variants. Overloaded operators let programmers implement algorithms in syntax very similar to the mathematical formulations, which reduces the chance of programmer error and leads to code that is easier to understand and modify. Still, achieving the full performance available in optimized, compiled languages is essential for scientific applications.

In this article, we'll reinterpret the expression templates for matrix computation put forward in previous columns[1–3] in light of functional programming ideas. Existing expression template libraries frequently focus on componentwise operations such as addition at the expense of matrix multiplication. The ideas presented here are implemented in a C++ library called LLANO (for Lazy Linear Algebraic Numerical Objects).

A Functional Approach to Componentwise Operations

Mathematically, matrix addition is defined as mapping from pairs of matrices back into themselves. Making a similar definition for matrix addition, however, gives poor performance. To see this, suppose that for some matrix class we define

```
Matrix operator+(const Matrix& A,
      const Matrix& B) {
   if(A.getNumRows() != B.getNumRows() ||
      A.getNumCols() != B.getNumCols()) {
     cerr << "nonconforming matrices... +"
       << endl;
     exit(-1);
   }
   Matrix new_mat(A.getNumRows(),
       A.getNumCols());
   for (int i=0;i<A.getNumRows();i++) {
   for (int j=0;j<A.getNumCols();j++) {
      new_mat(i,j) = A(i,j) + B(i,j);
    }
   }
   return new_mat;
}
```

Consider now the behavior of the code

```
A = B + C;
```

where a temporary matrix is created on the right-hand side, which stores the value of `B+C`. On assignment, these values must be copied into `A`. This causes an extra allocation–deallocation pair; it also copies n^2 values relative to a simple loop such as that imbedded in the definition of +.

The situation worsens as we add additional operands. In the code

```
A = B + C + D + E;
```

three temporaries are created. The cost relative to a hand-coded loop diverges quickly with an expression's complexity.

Instead of the mathematical definition, let's engage in a bit of functional programming and define addition on $n \times n$ matrices as

$$+ : M \times M \to \{f : [1, n] \times [1, n] \to \Re\}$$

according to the rule

$$A + B \to +_{A,B},$$

where $+_{A,B}(i, j) = A(i, j) + B(i, j)$. Thus, addition now returns a function that we can evaluate to compute the underlying matrix sum's entries. If we (formally) apply this definition to

Dave's Sideshow

Remembering the CM-5

A few months ago, I was sorting through a bunch of papers in my office (in a feeble attempt to get organized) and came across an old photograph of the Connection Machine 5 at Los Alamos. One of my students then remarked, "Wow! What is that?" Shortly thereafter, it struck me that over 10 years had passed since my first exposure to parallel computing and the CM-5. It also hit me that I probably could get something of roughly the same computing power for a few thousand dollars at the corner store thanks to Moore's law. Nevertheless, this got me thinking about how much fun it was to use the old parallel machines.

The CM-5 really was an interesting machine for those who used it. For one thing, it looked really cool—large, black, and with thousands of red blinking lights. I mean, who wouldn't want to have a machine that looked like that? More importantly, no one really quite knew the most effective way to program it. It seemed like many of our early programs were adapted from short examples and experimental use of undocumented features found in header files—after all, there was no such thing as MPI. Even now, I fondly look back at the insanity of `printf` debugging on 256 processors. Or, the two months of head-throbbing pain I endured to learn that one should never, ever, ever, ever use asynchronous message passing for anything, no matter how good the idea might sound at the time. Or, the one CPU board installed in the machine that seemed to work with everyone else's code except for mine. And who could forget the occasional email from the computer center reporting that "creeping death" had been detected—a pathological "blue screen of death" effect that took about a half an hour to reach its full strength? If you're going to crash, you might as well crash hard.

Looking back, I still chuckle when I think of the TV crew that came to the lab to do a story about supercomputers (along with some cool futuristic shots of the CM-5 in action). At the time, no one realized that the TV story was really a lead-in for a discussion in which Pat Robertson compared supercomputers to the biblical "beast." Well, what can you say? A machine like the CM-5 certainly had personality. Several years later, I visited a colleague at Sandia and noticed that he had a manual for the Intel Paragon mounted on the wall of his office like a little trophy—and it was shot full of bullet holes! Apparently the "beast" had a few friends.

The days of special-purpose parallel machines seem long gone now. Although it is amazing to see what people are doing with clusters, it still doesn't change the fact they're mostly just a bunch of PCs. In many respects, I long for the days of blinking lights, creeping death, and programming problems of, well, biblical magnitude.

"It's a Memory Problem"

Lately, I've been teaching our operating systems course. For the most part, this is a head-exploding exercise in C pointer manipulation in which I spend eight weeks telling students that "it's a memory problem" whenever they have trouble. Of course, I wouldn't lie—almost all C programming problems are, in fact, memory problems. As an instructor, I can only hope that students get all of this out of their systems in this one course. Regrettably, the ever increasing number of worms, viruses, and buffer overload attacks seems to indicate that students aren't the only ones suffering from memory problems.

Of Interest

If you're interested in scientific software components, take a look at the Babel project (www.llnl.gov/CASC/components/babel.html). On the other hand, if you're too busy chasing down bugs to even think about components, you might look at Roundup (http://roundup.sf.net), a nice issue tracker that has been gaining in popularity. Last, but not least, the O'Reilly Open Source Software Convention is rapidly approaching. Details are available at http://conferences.oreillynet.com/os2003.

a case in which one of the operands is itself a sum, we get

$$A + B + C \to +_{A,B} + C \to +_{+_{A,B},C}.$$

We can evaluate this function applied to (i, j) by repeated substitution to get

$$+_{+_{A,B},C}(i,j) = +_{A,B}(i,j) + C(i,j) = A(i,j) + B(i,j) + C(i,j).$$

Although C++ is not a functional language, several projects are underway to use its templating system to implement this approach. Among these are FC++ (www.cc.gatech.edu/yannis/fc++), FACT! (www.kfa-juelich.de/zam/FACT/start/index.html), and the Boost lambda library (www.boost.org/libs/lambda/doc/index.html).

If we define assignment to perform a loop over all entries and evaluate the right-hand side, we create a mathematical

formalism that models matrix addition without creating temporaries. To translate this into C++, we need two things: one, a polymorphism that lets the + operator work on matrices and sums of matrices, and two, the ability to substitute the body of $+_{A,B}(i,j)$ to squeeze out extra function calls and produce code similar to a hand-coded loop. This latter ability is also known as *function inlining*.

A natural first choice for an object-oriented programmer would be a class hierarchy in which both matrices and sums are derived from some common base type with a pure virtual indexing operator. The matrix sum class would contain (references to) elements of the base class. The loss of type information in this genericity prevents the compiler from inlining the index functions, so all indexing must be resolved via the virtual function table at runtime. This leads to problems comparable to creating or destroying all the temporaries.

Of course, C++ provides another type of genericity that preserves type information: the template system. In addition to a regular dense matrix class, we can introduce a class for matrix sums in which the operands are generic (the addition operator is likewise generic). Then, an appropriate class of matrix sums is instantiated for each pair of types of operands:

```
template<class U,class V>
class MatrixSum {
public:
       MatrixSum(const U&u, const V&v) :
         u(u), v(v) {}
       int getNumRows() const {return u.
          return u.getNumRows();
       int getNumCols() const {return u.
          return u.getNumCols();
       double operator()(int i, int j) const
          {return u(i,j) + v(i,j);}
private:
       const U&u;
       const V&v;
}

template<class U,class V>
MatrixSum<U,V> operator+(const U&u, const
V&v) {
       // check for conforming
       // dimensions...and then
     return Matrix Sum<U,V>(u,v);
}
```

In the code `D = A + B + C;`, several things happen. First, a `MatrixSum<Matrix,Matrix>` object is created that represents `A+B`. This is then combined into a `MatrixSum<MatrixSum<Matrix,Matrix>,Matrix>` object that represents the entire right-hand side. The assignment operator forces the computation via a loop over rows and columns, indexing the top-level matrix sum and assigning the result to the corresponding entry of `D`. However, because the exact type is known, this indexing operator could be inlined, resulting in the sum of the i, j entry of another matrix sum (`A+B`) with the i, j entry of `C`. Again, knowledge of type information lets the compiler inline this matrix sum's indexing, leading to a loop over rows and columns such that the inner computation is just `D(i,j) = A(i,j)+B(i,j)+C(i,j);`.

However, each of the indexing functions into matrices can be inlined as well, meaning that the compiled code is reading from and writing directly to the underlying arrays and not going through any intermediate functions. Hence, the total cost of the operator overloaded code is the cost of the standard C code plus the cost of instantiating and destroying the expression objects (this is $\mathcal{O}(1)$ and, hence, asymptotically negligible).

Implementing this `MatrixSum<U,V>` class, which computes an entry of the sum upon being indexed, mimics the functional programming idea nicely. In another language, we could use actual functional programming, but for our purposes, the functional inspiration translates into C++ satisfactorily. Note that once this is understood for addition, all other componentwise operations follow. This is the driving force behind, for example, the highly developed Blitz++ library.[4]

Incorporating Matrix Multiplication

Although this formalism makes the idea of expression templates clearer, we still need an expression template system capable of efficiently supporting operations such as

```
Z = (A + B) * C - D.transpose() * E;
```

where `*` is not componentwise multiplication but the actual matrix product. Few matrix expression template packages have addressed this. For example, Blitz++ originally did componentwise operator overloading but handled matrix multiplication through a simple `matmul` routine. Now, it supports actual matrix multiplication via tensor operations rather than through a multiplication operator.

Deferring `*` in the same manner as `+` also can have serious consequences. First, it necessarily uses the *ijk* ordering to compute each entry of the matrix product. This produces the worst possible memory access patterns when column-

oriented storage is used.[5] Second, deferring operations leads to an increased operation count for many problems. Evaluating the product `A * B * C` by this technique, for example, is an $\mathcal{O}(n^4)$ process. Finally, libraries such as BLAS (Basic Linear Algebra Subroutines) require the actual data for their arguments, not code for evaluating those operands. This approach thus precludes the use of such libraries.

Although the creation of temporaries in componentwise operations destroys efficiency (the extra memory traffic is $\mathcal{O}(n^2)$), the extra memory traffic associated with multiplication is of lower order. As we saw earlier, the cost of avoiding temporaries by deferring evaluation can be much larger than creating them. So, the question arises of how to automate the creation and destruction of temporaries. For large matrices, the approach need not be optimal because it's a lower-order term. However, many matrix applications use computations on medium-size matrices, so we really should optimize this regime.

Let's use the following approach for simplicity's sake. Assign a mutable cache to each expression object (sums, products, differences, and so on). When the expression appears as an operand for multiplication, the multiplication allocates the cache, evaluates the expression into the cache, and passes it to the library routine. The expression object destructor then deallocates this cache. Likewise, when multiplication appears as an operand, it is evaluated into its cache before the expression is evaluated.

The introduction of local caching requires new methods and properties for each expression class. Each class needs a `fill(int)` method that takes a message indicating the kind of forcing applied to it. If the message is "strong enough," then the object responds by filling its cache with the value of the expression it represents. The method `Matrix::fill(int)` is empty, regardless of the argument. The `fill` method for componentwise operations such as addition pass when called by other componentwise operations, but evaluate themselves when called by multiplication. Matrix products always fill their cache when they are forced.

In addition to this `fill` method, the `fillInto()` method writes the expression's results into a target matrix. This lets us evaluate "in place" by making = a wrapper to the underlying `fillInto()` method. We have two such `fillInto()` methods to allow for the operation $A \leftarrow B$ and $A \leftarrow a * A + b * B$, which is useful in implementing update operators such as `+=`.

As an example, the `fillInto(Matrix&)` method of the MatrixSum class is

```
template{class U, class V>
void MatrixSum<U,V>::fillInto(Matrix&A) const {
  // evaluate arguments if necessary
  u.fill(MU);
  v.fill(MU);
  // evaluate
  for (int j=0;j<A.getNumCols();j++) {
     for (int i=0;i<A.getNumRows();i++) {
        A(i,j) = operator()(i,j);
     }
  }
}
```

Here, messages are passed to the sum's arguments; they fill their cache only if they are products. Otherwise, these calls are empty, and we maintain the inlining discussed earlier. Now, assignment is implemented as

```
template<class U>
Matrix& Matrix::operator=(const U&u) {
  //dimension check here
  u.fillInto(*this);
  return (*this);
}
```

This abstraction of separating assignment from computation by the layer of `fillInto` gives us the freedom to evaluate componentwise operations via a loop with inlined indexing and also to pass the pointer to a matrix's storage into BLAS to evaluate multiplication.

Let's look at the process of evaluating `D = (A+B)*C`. First, the `MatrixSum` object is created, followed by the `MatrixProduct` object (with the first argument being a matrix sum). The assignment operator begins a chain of evaluation, calling first the `fillInto()` method of the `MatrixProduct` object to write into the buffer of `D`. Then, the `fillInto()` method calls the arguments' `fill()` method. The first argument is a matrix sum. Because products force sums to be evaluated, the matrix sum's cache is allocated and filled with the result. The second argument of the product's `fill()` method does nothing because `fill()` is always a pass for matrices. After the arguments are filled, the caches and buffers are passed to the library routine to execute the multiplication into the storage for `D`. At the destruction of the `MatrixProduct`, its arguments are destroyed, deallocating their caches.

More generally, we generate an expression tree; calling the `fillInto()` method of the top node leads to a tree recursion of sending messages down the branches. These mes-

Table 1. Timing results for `3*A–B+C`, *n* x *n* matrices.

m	*n*	LLANO	Hand-coded
25	25	0.27	0.13
50	50	0.47	0.32
100	100	1.06	0.96
200	200	1.32	1.29
400	400	1.30	1.32
800	800	1.34	1.28

Table 2. Timing results (in seconds) for `D = (A + B) * C + A * B + C;`.

m	*n*	LLANO	Hand-coded
25	25	0.1825	0.1575
50	50	0.3675	0.3475
100	100	1.21	1.1675
200	200	1.88	1.8575
400	400	3.2375	3.23
800	800	5.795	5.85

sages either defer or force the evaluation of each node, depending on the type of operations.

LLANO

All these ideas are implemented in LLANO (available online at http://people.cs.uchicago.edu/~kirby/LLANO.html). Table 1 shows some timing results comparing LLANO to hand-coded C implementations of the same operations. The table shows that the cost of the operator template system in LLANO does not interfere with the code's asymptotic performance. For componentwise operations at least, Blitz++ produces similar convergence to the C code. All results are computed on a Macintosh iBook with a 600-MHz G3 processor and 640 Mbytes of RAM. The code is compiled with g++ version 2.95 using -O3 optimization. Matrix multiplication is handled through the Atlas-generated CBLAS library routine `cblas_dgemm`.

Now consider the LLANO statement

```
D = (A + B) * C + A * B + C;
```

where `A`, `B`, `C`, and `D` are all $n \times n$. Computing this without overloaded operators requires several steps. First, `A + B` must be stored in a temporary, which must then be multiplied by `C` and stored in another temporary. Then, `A * B` must be stored into a third temporary. Finally, the second and third temporaries must be added to a loop. Table 2 compares LLANO code against C code that manually calls the BLAS routines and handles temporary storage and reading or writing.

An elegant and efficient implementation of operator overloading shows great promise in allowing programmers to write code at a natural level, yet still get good performance. LLANO shows that memory management can even automatically include temporaries necessary for matrix multiplication.

However, C++ (at least in its current state) still produces several obstacles as a language for this endeavor. Although the compiled code's performance no longer seems an impediment to using C++, the lack of support among compilers for the `export` keyword is disturbing. Including a header file separately in each file of a large code leads to platform-dependent issues regarding multiply defined symbols. Also, template syntax is far from natural in some circumstances.

Matlab's success has demonstrated that users are willing to learn a new system provided it supplies sufficient power and expressivity. Perhaps the time is right to explore other compiled languages with functional features for making code more elegant. This could be a step toward the goal of combining high performance and ease of use (both in terms of natural syntax and portability) into a single programming environment.

Acknowledgments

I thank Stuart Kurtz for many helpful discussions. The Climate Systems Center at the University of Chicago supported this work under NSF grant AP/ITR 0121028.

References

1. S.W. Haney, "Is C++ Fast Enough for Scientific Computing?" *Computers in Physics*, vol. 8, no. 6, 1994, pp. 690–695.
2. S.W. Haney, "Beating the Abstraction Penalty in C++ Using Expression Templates," *Computers in Physics*, vol. 10, no. 6, 1996, pp. 552–557.
3. A.D. Robison, "C++ Gets Faster for Scientific Computing?" *Computers in Physics*, vol. 10, no. 6, 1996, pp. 458–462.
4. T. Veldhuizen, "Arrays in Blitz++," *2nd Int'l Scientific Computing in Object Oriented Parallel Environments* (ISCOPE '98), Springer-Verlag, 1998; www.oonumerics.org/blitz.
5. G. Golub and C. van Loan, *Matrix Computations*, Johns Hopkins Univ. Press, 1996.

Robert C. Kirby is an assistant professor of computer science at the University of Chicago. His technical interests include numerical analysis, scientific computing, and mathematical software. He received his PhD in computational and applied mathematics from the University of Texas, Austin. Contact him at the Dept. of Computer Science, Univ. of Chicago, 1100 E. 58th St., Chicago, IL 60637; kirby@cs.uchicago.edu.