Software Engineering for Peak Performance

Jeremy Siek and Andrew Lumsdaine

OVER THE PAST few years there have been growing reports of how C++ can be as fast as Fortran on numerical codes, due to the combination of clever template techniques[1] and better optimizing compilers.[2] In this article, we continue this exploration into high performance C++ by delving into the area of numerical linear algebra libraries. We show that one can achieve performance on a par with the vendor-tuned basic linear algebra subroutines (BLAS) even on large out-of-cache problems. In addition, we show how to engineer performance optimizations in a portable and concise fashion using template techniques, which results in a big gain from the point of view of software development and maintenance.

DigitalVision

The area of linear algebra libraries has always been a stronghold for Fortran and assembly programming. Each line of code is tuned for the last megaflop and each algorithm is contorted to take maximum advantage of the cache. For high profile benchmarks such as Linpack,[3] man-years are spent optimizing and porting for each new generation of supercomputer. Common wisdom says that it is impossible to write such high performance codes in C++. We demonstrate that not only is it possible to achieve the same high performance with a C++ implementation, but that in several respects the use of C++ makes the library development easier.

AUTOMATING OPTIMIZATION Optimizing linear algebra libraries for modern RISC processors is a well-understood, though complex, process. Each of the major computer vendors provides tuned versions of the BLAS,[4] and there are several open-source packages that automatically tune for various architectures (PHiPAC[5] and ATLAS[6]). The move towards automation is critical to reducing the cost of development and maintenance of these high performance codes.

The approach that PHiPAC and ATLAS use to automate the process is to have a set of scripts that generate specialized C code during library installation. This method is effective but difficult to develop and maintain. The numerical algorithms are controlled in an indirect way through the code generation scripts, making it hard to modify or extend them. The approach that we take is to use the powerful compile-time computation mechanisms inherent in C++ templates to produce specialized versions of the algorithms. Such template techniques look strange at first, but they are actually a straightforward way to express certain optimizations. We used these techniques in construction of the Matrix Template Library (MTL),[7] which is a library of generic components for linear algebra.

OPTIMIZATION TECHNIQUES Modern RISC processors are much like thoroughbred racehorses. These processors are built to go very fast, but only under very specific conditions (which happen to match up well with the popular benchmarks). If the particular algorithm or code that you are writing does not meet these conditions, then you can expect to see only a very small fraction of peak performance—often less than 10%!

Modern RISC processors are designed to go fast when programs exhibit good data locality. There are two kinds of locality: temporal and spatial.[8] Temporal locality is just the reuse of the same piece of data in a program at different times, such as data reused in each iteration of a loop. Spatial locality is characterized by subsequent accesses to memory that are nearby one another. An algorithm that traverses an array sequentially from beginning to end is a good example of this.

The hardware features that take advantage of this are primarily the registers and cache, which reduce the time to access memory from hundreds of cycles to just a few. Because of the high speeds of today's CPUs in comparison to their speed of memory access (and this gap is growing wider every year!), the efficiency of most code depends much less on how many multiplications or additions are performed, but on how many loads and stores are performed, and on how much locality the algorithm exhibits.

Understanding locality is very important for optimizing linear algebra kernels. Let us take a quick look at an example. The following loops perform a matrix vector multiply (y = Ax + y):

```
for (i = 0; i < M; ++i )
  for (j = 0; j < N; ++j )
    y[i] = A[i][j] * x[j] + y[i];
```

The reference to y[i] has temporal locality with respect to the inner loop. The references to A[i][j] and x[j] both have spatial locality. It is important to keep in mind the kind of assembly instructions that correspond to the C++ expressions. Typically, array accesses correspond to a load or store instruction, and operations on temporaries or local variables correspond to register instructions. Using a little knowledge about locality, we can rewrite the above example to reduce the number of loads and stores.

```
for (i = 0; i < M; i += 2 ) {
  y_i = y[i];
  y_ip1 = y[i+1];
  for (j = 0; j < N; ++j ) {
    x_j = x[j];
    y_i = A[i][j] * x_j + y_i;
    y_ip1 = A[i+1][j] * x_j + y_ip1;
  }
  y[i] = y_i;
  y[i+1] = y_ip1;
}
```

The first thing to notice is that the reference to y[i] has been completely removed from the inner loop. This means that the number of loads and stores to and from array y has been reduced from M*N to just M. In addition, the outer loop has been unrolled (note how variable i is incremented by 2 instead of 1), allowing the reference x[j] to be reused twice (instead of the single use per iteration in the original loop). These changes show how code can be transformed to better reuse registers. Similar kinds of transformations called blocking or tiling can be applied at the cache level as well. Modern optimizing compilers attempt to perform these kinds of optimizations to varying degrees of success (they are very sensitive to the exact nature of the loops). In situations where extremely high performance must be guaranteed, these kinds of optimizations are done by hand.

The other major performance-enhancing feature of modern RISC processors is the CPU pipeline. This allows the processor to take advantage of low-level parallelism in your program, executing as many instructions as possible at the same time. Pipelines are also finicky creatures, and tend to stall if they are not pampered. They particularly dislike such things as branches and data-dependencies, and in general like their instructions to be carefully ordered for easier consumption. Fortunately, modern optimizing compilers are quite adept at tailoring the instruction stream to pipelined CPUs (the unrolling optimizations previously discussed also help pipeline behavior), which relieves the library writer from the trouble of fine-tuned instruction scheduling.

PORTABLE HIGH PERFORMANCE One of the difficulties in optimizing for high performance is that each processor is different. They have different numbers of registers, different cache sizes, different memory access times, etc. Specializing for numbers of registers and pipeline characteristics is a serious difficulty because this entails changes to the source code of all the inner loops and recompilation. For example, one might need to unroll by three to achieve optimum performance in the previous example, instead of unrolling by two. This is why PHiPAC and ATLAS use scripts to generate C code. Varying degrees of unrolling cannot be expressed in C or Fortran. We will now describe how this can be done in C++, and show how we have used this capability in constructing our linear algebra libraries.

AN INTRODUCTION TO THE BLAIS The Basic Linear Algebra Instruction Set (BLAIS) is a small library that we have developed to aid in the construction of the MTL. The BLAIS takes advantage of the powerful compile-time computation features of C++ templates[9] to provide kernels with parameterized unrolling factors. We will start with a simplified example of how BLAIS can be used. The following is the unrolled matrix-vector algorithm from the previous example, rewritten using the BLAIS:

```
in mtl/blais_config.h:
// unrolling factors for blocked matrix-vector multiply
#define MV_UNROLL_M 2
#define MV_UNROLL_N 1

in mtl/mtl_algo.h:
#include <mtl/blais_config.h>
#include <mtl/blais.h>

...
template <class Matrix, class VectorX, class VectorY>
void blocked_mult(const Matrix& A, const VectorX& x, VectorY& y )
```

```
{
 const int M = A.nrows( ), N = A.ncols( );
 blocked_matrix<Matrix,MV_UNROLL_M,MV_UNROLL_N> BA(A );
 blocked_vector<VectorY,MV_UNROLL_M> bY(y );
 typename blocked_vector<VectorY,MV_UNROLL_M>::reference bY_i;
 blocked_vector<VectorX,MV_UNROLL_N> bX(x );
 bM = M / MV_UNROLL_M;

 for (i = 0; i < bM; ++i ) {
  bY_i = bY[i];
  for (j = 0; j < N; ++j )
   blais::mult<MV_UNROLL_M,MV_UNROLL_N>(BA(i,j ), bX[j], bY_i );
  bY[i] = bY_i;
 }
}
```

The blocked matrix BA is a matrix of tiny (2x1) submatrices and the blocked vectors bY and bX are vectors of tiny subvectors (note that the blocked objects are just a "view" to the original data—no extra allocation is done here). Inside the loops there is a call to the blais::mult<MV_UNROLL_M,MV_UNROLL_N>( ) operation. This multiplies the tiny submatrix by the tiny vectors. From this example we can see the two advantages of the BLAIS: The unrolling and blocking is easier to express, and more importantly, the amount of unrolling has been parameterized. This means the code does not have to change from architecture to architecture!

TEMPLATE METAPROGRAMMING In this section we focus on how blais::mult( ) is implemented. There are a few simple layers of functions that go into the implementation of blais::mult( ) (and the other BLAIS operations). The first layer is a set of generic algorithms similar to the STL that have been customized to work on fixed (at complie time) sized ranges. We call this layer the Fixed Algorithm Size Template (FAST) library.

One of the STL algorithms that can be used to perform linear algebra operations is transform( ). In the following example the array x is added to the array y:

```
int x[4] = {1,3,5,4}, y[4] = {2,1,2,5};
std::transform(x, x + 4, y, y, std::plus<int>( ) );
```

The transform( ) algorithm consists of a single loop that applies the binary operation to the two input iterators and stores the result in the output iterator. To create an unrolled version of this algorithm, we use a recursive template function with an integer template argument to control the recursion. The purpose of the count object is to carry the template argument N. The template function is specialized to stop the recursion for the case when N equals 0.

```
namespace fast {
 // a helper class to carry "N"
 template <int N> struct count { };

 // The general case
 template <int N, class InIter1, class InIter2,
           class OutIter, class BinOp>
 OutIter transform(InIter1 first1, count<N>, InIter2 first2,
                   OutIter result, BinOp binary_op ) {
  *result = binary_op (*first1, *first2 );
  return transform(++first1, count<N-1>( ), ++first2,
                   ++result, binary_op );
 }
 // The N == 0 case to stop template recursion
 template <class InIter1, class InIter2,
           class OutIter, class BinOp>
 OutIter transform(InIter1 first1, count<0>, InIter2 first2,
                   OutIter result, BinOp binary_op ){
  return result;
 }
}
```

If we use this FAST version of the algorithm and compile with optimizations turned on, the resulting code contains no loops and is highly efficient for tiny vectors.

```
fast::transform(x, fast::count<4>( ), y, y, std::plus<int>( ) );

// the compiler expands transform( ) to:
// y[0] = x[0] + y[0];
// y[1] = x[1] + y[1];
// y[2] = x[2] + y[2];
// y[3] = x[3] + y[3];
```

Ultimately we wish to express register-level blocking for linear algebra kernels, which equates to performing linear algebra operations on tiny matrices and vectors (typically on the order of 4x4 or less). On top of the FAST library, we construct the BLAIS operations that operate on these tiny matrices and vectors. The following class defines the blais::add( ) operation for tiny vectors in terms of the FAST transform( ):

```
namespace blais {
 template <int N> struct add {
  template <class VectorX, class VectorY>
  add(const VectorX& x, VectorY& y ) {
   typedef typename VectorX::value_type T;
   fast::transform(x.begin( ), fast::count<N>( ), y.begin( ),
                   y.begin( ),std::plus<T>( ) );
  }
 };
}
```

Using the blais::add( ) operation, we can then construct a matrix-vector multiply operation. This version computes a linear combination (using vector addition) of the columns of a column-major matrix:

```
namespace blais {
 template <int M, int N>
 struct mult {
  template <class ColMatrix, class VectorX, class VectorY>
  mult(const ColMatrix& A, const VectorX& x, VectorY& y ) {
   recursive_mult<M,N>(A.begin( ), x.begin( ), y );
  }
 };
 template <int M, int N>
 struct recursive_mult {
  template <class Iter2D, class IterX, class VectorY>
  recursive_mult(Iter2D Ai, IterX xi, VectorY& y ) {
   add<M>(scale(*Ai, *xi ), y ); recursive_mult<M, N-1>(++Ai, ++xi, y );
  }
 };
```

```
  // specialize recursive_mult for N == 0 to stop recursion...
}
```

The call to the scale( ) function returns an adapter that causes the column to be scaled (multiplied) during the addition by an element from the array x. The implementations of the other BLAIS operations, such as the matrix-matrix product, are very similar.

The Iter2D is an iterator for a 2D STL-style container. It is an object with the same interface as std::vector< vector<T> >. MTL uses the 2D-container interface for its matrix classes. Dereferencing an Iter2D gives a column of the matrix. A column then has its own begin( ) and end( ) functions and associated iterators for accessing its individual elements.

MAYFLIES AND LIGHTWEIGHT INTERFACES With all the layers of functions and objects, one might think that there is so much overhead in the BLAIS and FAST libraries that their use would make linear algebra codes slower, not faster. Luckily, with a good optimizing C++ compiler, and a few implementation tricks, the overhead simply evaporates at compile time.

First, the BLAIS and FAST libraries use template functions (never virtual functions), which good C++ compilers will inline. Consequently, the BLAIS and FAST libraries are completely flattened out. Second, we rely on the compiler to properly optimize small objects (this is known as lightweight object optimization[10] or scalar replacement of aggregates[11]). This optimization replaces small objects such as the tiny matrices and vectors used in BLAIS with the components with which they are made up. This allows other optimizations to work in the presence of objects (such as register allocation and copy propagation). To enable lightweight object optimization, one must follow a particular coding style when creating tiny objects, which we name the Mayfly pattern.[12] Mayflies are stack-allocated objects and typically consist of just a pointer to some data and an integer index. We have found that mayflies are extremely useful for creating lightweight generic interfaces for such things as array-based graphs, trees, and heaps.

Here is an example of one of the mayfly classes used in BLAIS, the vector_view class. This class plays the role of a subvector within a blocked vector. The only member field of the class is the pointer _data, which makes this a very lightweight object. The member functions of this class are small and nonrecursive, which allows a good compiler to inline all of them.

```
template <class T, int N>
class vector_view {
public:
  typedef T* iterator;
  typedef T& reference;
  ...
  vector_view(T* d ) : _data(d ) { }
  reference operator[](size_type i ) { return _data[i]; }
  iterator begin( ) { return _data; }
  iterator end( ) { return _data + N; }
  ...
protected:
  T* _data;
};
```

To really understand the "mayfly" or short-lived aspect of vector_view objects, we take a look at how a vector_view is created. An excerpt from the blocked_vector follows. The value_type and reference type of blocked_vector is a subvector, or an instance of the vector_view class. However, a blocked_vector does not actually contain any vector_view objects. Instead, the vector_view objects are created on demand. Note how the operator[] creates a new temporary vector_view object and returns it after initializing the data pointer. The lifetime of the vector_view object therefore typically only extends to the expression in which it is used—extremely ephemeral!

```
template <class T, int B> // B is the blocking factor
class blocked_vector {
public:
  typedef vector_view<T, B> reference;
  ...
  reference operator[](int i ) {
    return reference(_start + i * B );
  }
  ...
};
```

The advantage of using mayflies is that you can provide the appearance of a nice object-oriented interface with all the abstraction without paying in time or space overheads. The temporary objects and extra functions get optimized away!

PERFORMANCE EXPERIMENTS Now we present a set of experiments that provide performance results comparing MTL, using the BLAIS and FAST sublibraries, with other available libraries (both public domain and vendor supplied). The algorithms timed were the dense matrix-matrix multiplication, the dense matrix-vector multiplication, and the sparse matrix-vector multiplication.

Figure 1 shows the dense matrix-matrix product performance for MTL, Fortran BLAS, the Sun Performance Library, TNT14, and ATLAS, all obtained on a Sun UltraSPARC 170E. The experiment shows that the MTL can compete with vendor-tuned libraries (on an algorithm that tends to get extra attention due to benchmarking). The MTL and TNT executables

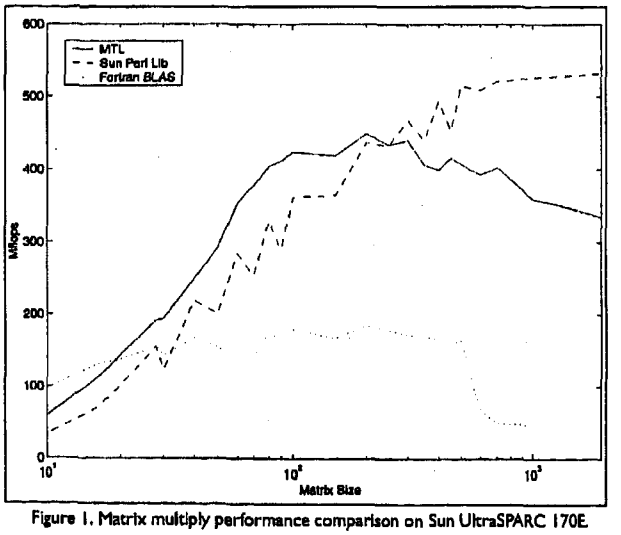

Figure 1. Matrix multiply performance comparison on Sun UltraSPARC 170E.

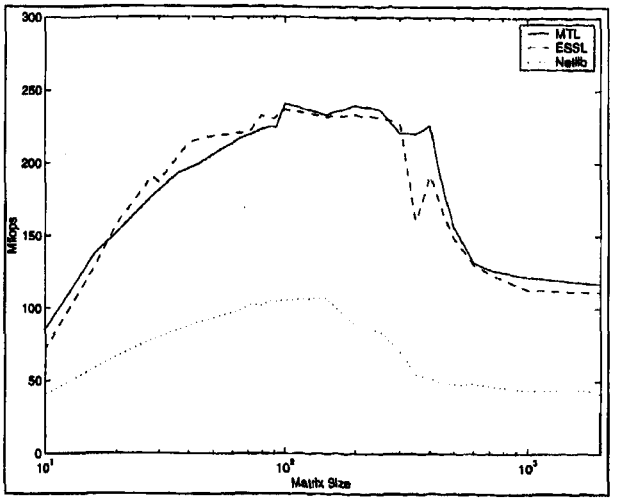

Figure 2. Matrix multiply performance comparison on IBM RS/6000.

were compiled using KCC, in conjunction with the Solaris C compiler. ATLAS was compiled with the Solaris C compiler, and the Fortran BLAS (obtained from Netlib[13]) were compiled with the Solaris Fortran 77 compiler. All possible compiler optimization flags were used in all cases. The cache was cleared between each trial of the experiment. To demonstrate portability across different architectures and compilers, Figure 2 also compares the performance of MTL with ESSL[14] on an IBM RS/6000 590. In this case, the MTL executable was compiled with the KCC and IBM xlc compilers.

The presence (and absence) of optimization techniques in the different programs can readily be seen in Figure 1 and manifests itself quite strongly as a function of matrix size. In the region from N=10 to N=256, performance is dominated by register usage and pipeline performance. "Unroll and jam" techniques[15] are used to attain high levels of performance in this region. In the region from 256 to approximately 1024, performance is dominated by data locality. Loop blocking for cache usage is used here to attain high performance levels. Finally, for matrix sizes larger than approximately N=1024, performance can be affected by conflict misses in the cache—notice how the results for the Fortran BLAS fall precipitously at this point. To attain good performance in the face of conflict misses in low associativity caches, block-copy techniques[16] are used. Note that performance effects are cumulative. For instance, the Fortran BLAS do not use any of the techniques listed for performance enhancement. Therefore, performance is poor initially and continues to degrade as different effects come into play.

AVAILABILITY The Matrix Template Library (of which BLAIS and FAST is a part) is available for download at http://www.lsc.nd.edu/research/mtl.

CONCLUSION Thanks to some significant advances in C++ optimizing compilers, it is now possible to construct high-level software that is also high-performance. In addition, portability issues for ultrahigh-performance software can be addressed with compile-time specialization using the C++ template facilities. We feel that this approach is much more straightforward than others (such as code generation scripts) and results in significantly reduced development and maintenance effort.

In general, we feel that there is great potential in the compile-time computation capabilities of C++, of which our work just scratches the surface. Several other projects, such as POOMA[17] and Blitz++,[18] are also exploring the domain of "active libraries," which use C++ templates to be highly adaptive and self-optimizing. ❮

References

1. Veldhuizen, T. "Rapid Linear Algebra in C++," *Dr. Dobb's Journal*, Aug. 1996.
2. Robison, A. "C++ Gets Faster for Scientific Computing," *Computers in Physics*, 10(5): 458–462, Sep./Oct. 1996.
3. Dongarra, J. et al. *LINPACK Users Guide*, SIAM, Philadelphia, PA, 1978.
4. Dongarra, J. et al. "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, 16(1): 1–17, 1990.
5. Bilmes, J. et al. "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology," *Technical Report CS–96–326*, University of Tennessee, May 1996. Also available as LAPACK working note 111.
6. Whaley, R. C. and J. Dongarra. "Automatically Tuned Linear Algebra Software (ATLAS)," *Technical Report*, University of Tennessee and Oak Ridge National Laboratory, 1997.
7. Siek, J. *A Modern Framework for Portable High Performance Numerical Linear Algebra*, Master's Thesis, Univ. of Notre Dame, 1999.
8. Hennessy, J. L. and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, CA, 1996.
9. Veldhuizen, T. "Using C++ Template Metaprograms," *C++ Report*, 7(4): 36–43, May 1995.
10. Robison, A. "The Abstraction Penalty for Small Objects in C++." *POOMA* 1996, http://www.acl.lanl.gov/Pooma96/abstracts/robison.html.
11. Muchnick, S. *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, CA, 1997.
12. Siek, J. and Lumsdaine, A. *The Mayfly Pattern. Pattern Languages of Program Design* (PloP), 1999.
13. Netlib BLAS http://www.netlib.org/blas/index.html.
14. IBM. *Engineering and Scientific Subroutine Library, Guide and Reference*, 2nd ed., 1992.
15. Carr, S. and Y. Guan. "Unroll-and-Jam Using Uniformly Generated Sets," in *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO–97)*, pp. 349–357, Los Alamitos, IEEE Computer Society, Dec. 1–3, 1997.
16. Lam, M. S., E. E Rothberg, and M. E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms," in *ASPLOS IV*, Apr. 1991.
17. Advanced Computing Laboratory, *POOMA*, http://www.acl.lanl.gov/pooma.
18. Veldhuizen, T. *Blitz++*, http://oonumerics.org/blitz/index.html.