A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library *

Jeremy G. Siek Andrew Lumsdaine

Laboratory for Scientific Computing,
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN 46556,
{jsiek,lums}@lsc.nd.edu,
WWW home page: http://www.cse.nd.edu/~lsc

Abstract. We introduce a collection of high performance kernels for basic linear algebra. The kernels encapsulate small fixed size computations in order to provide building blocks for numerical libraries in C++. The sizes are templated parameters of the kernels, so they can be easily configured to a specific architecture for portability. In this way the BLAIS delivers the power of such code generation systems as PHiPAC [1] and ATLAS [8]. BLAIS has a simple and elegant interface, so that one can write flexible-sized block algorithms without the complications of a code generation system. The BLAIS are implemented on the Fixed Algorithm Size Template (FAST) Library which we also introduce in this paper. The FAST routines provide equivalent functionality to the algorithms in the Standard Template Library [7], but are tailored specifically for high performance kernels.

1 Introduction

The bane of portable high performance numerical linear algebra is the need to tailor key routines to specific execution environments. For example, to obtain high performance on a modern microprocessor, an algorithm must properly exploit the associated memory hierarchy and pipeline architecture (typically through careful loop blocking and structuring). Ideally, one would like to be able to express high performance algorithms in a portable fashion, but there is not enough expressiveness in languages such as C or Fortran to do so. Recent efforts (PHiPAC [1], ATLAS [8]) have resorted to going outside the language, i.e., to code generation systems in order to gain this kind of flexibility. In this paper we present the Basic Linear Algebra Instruction Set (BLAIS), a library specification that takes advantage of certain features of the C++ language to express high-performance loop structures that can be easily reconfigured for a particular architecture.

* This work was supported by NSF grants ASC94-22380 and CCR95-02710.

The BLAIS specification contains *fixed size* algorithms with functionality equivalent to that of the Level-1, Level-2, and Level-3 BLAS [6, 3, 2]. The BLAIS routines themselves are implemented using the Fixed Algorithm Size Template (FAST) library, which contains general purpose fixed-size algorithms equivalent in functionality to the generic algorithms in the Standard Template Library (STL) [7]. In the following sections, we describe the implementation of the FAST algorithms and then show how the BLAIS are constructed from them. Next, we demonstrate how the BLAIS can be used as high-level instructions (kernels) to construct a dense matrix matrix product. Finally, experimental results show that the performance obtained by our approach can equal and even exceed that of vendor-tuned libraries.

2 Fixed Algorithm Size Template (FAST) Library

The FAST Library includes generic algorithms such as `transform()`, `for_each()`, `inner_product()`, and `accumulate()` that are found in the STL. The interface closely follows that of the STL. All input is in the form of *iterators* (generalized pointers). The only difference is that the loop-end iterator is replaced by a *count template* object. The example shown in Fig. 1 demonstrates the use of both the STL and FAST versions of `transform()` to realize an `AXPY`-like operation ($y \leftarrow x + y$).

The `first1` and `last1` parameters are iterators for the first input container (indicating the beginning and end of the container, respectively). The `first2` parameter is an iterator indicating the beginning of the second input container. The `result` parameter is an iterator indicating the start of the output container. The `binary_op` parameter is a function object that combines the elements from the first and second input containers into the result containers.

The difference between the STL and FAST algorithms is that STL accommodates containers of arbitrary size, with the size being specified at run-time. FAST also works with containers of arbitrary size, but the size is fixed at compile time. In Fig. 2, we show how the FAST `transform()` routine is implemented. We use a tail-recursive algorithm to achieve complete unrolling — there is no actual loop in the FAST `transform()`. The template-recursive calls are inlined, resulting in a sequence of `N` copies of the inner loop statement. This technique (sometimes called *template metaprograms*) has been used to a large degree in the Blitz++ Library [10].

3 Basic Linear Algebra Instruction Set (BLAIS)

The BLAIS library is implemented directly on top of the FAST Library, as a thin layer that maps generic FAST algorithms into fixed-size mathematical operations. There is no added overhead in the layering because all the function calls are inlined. Using the FAST library allows the BLAIS routines to be expressed in a very simple and elegant fashion. The BLAIS library specification consists of fixed-size vector-vector, matrix-vector, and matrix-matrix routines.

```
int x[4] = {1,1,1,1}, y[4] = {2,2,2,2};

// STL
template <class InIter1, InIter2, OutIter, BinaryOp>
OutIter transform(InIter1 first1, InIter1 last1, InIter2 first2,
                  OutIter result, BinaryOp binary_op);

transform(x, x + 4, y, y, plus<int>());

// FAST
template <int N, class InIter1, class InIter2,
          class OutIter, class BinOp>
OutIter fast::transform(InIter1 first1, cnt<N>, InIter2 first2,
                        OutIter result, BinOp binary_op);

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

Fig. 1. Example usage of STL and FAST versions of `transform()`.

```
// The general case
template <int N, class InIter1, class InIter2,
          class OutIter, class BinOp>
inline OutIter
fast::transform (InIter1 first1, cnt<N>, InIter2 first2,
                 OutIter result, BinOp binary_op) {
  *result = binary_op (*first1, *first2);
  return transform(++first1, cnt<N-1>(), ++first2,
                   ++result, binary_op);
}

// The N = 0 case to stop template recursion
template <class InIter1, class InIter2, class OutIter, class BinOp>
inline OutIter
fast::transform (InIter1 first1, cnt<0>, InIter2 first2,
                 OutIter result, BinOp binary_op) { return result; }
```

Fig. 2. Definition of FAST `transform()`.

3.1 Vector-Vector Operations

Fig. 3 gives the implementation for the BLAIS `add()` routine. The FAST `transform()` algorithm is used to carry out the vector-vector addition as it was in the example above.

```
template <int N> struct vecvec::add {
  template <class Iter1, class Iter2> inline
  vecvec::add(Iter1 x, Iter2 y) {
    typedef typename iterator_traits<Iter1>::value_type T;
    fast::transform(x, cnt<N>(), y, y, plus<T>());
  }
};
```

Fig. 3. Definition of BLAIS `add()`.

Fig. 4 shows how the `add()` routine can be used. The comments on the right show the resulting code after the call to `add()` is inlined. Note that only one `add()` routine is required to provide any combination of scaling or striding. This is made possible through the use of the `scale_iterator` and `strided_iterator` *iterator adapters* (similar to the `reverse_iterator` adapter in the STL). The `scale_iterator` multiplies the value from `x` by `a` when the iterator is dereferenced within the `add()` routine. Any resulting overhead is removed by inlining and lightweight object optimizations [4]. The `scl(x, a)` call below automatically creates the proper `scale_iterator` out of `x` and `a`.

```
double x[4], y[4];                    // y[0] += a * x[0];
fill(x, x+4, 1); fill(y, y+4, 5);     // y[1] += a * x[1];
double a = 2;                         // y[2] += a * x[2];
vecvec::add<4>(scl(x, a), y);         // y[3] += a * x[3];
```

Fig. 4. Example use of BLAIS `add()`.

3.2 Matrix-Vector Operations

The BLAIS matrix-vector operations can be composed directly from BLAIS vector-vector routines and FAST algorithms. Fig. 5 shows the implementation of a matrix-vector multiplication (with a column-oriented matrix) using the BLAIS `add()`. The generic style of programming is extended to matrices in this algorithm. We view the matrix as a *container of containers*. For example, one could construct a matrix by composign STL vectors: `vector< vector<double> >`.

```
template <int M, int N> struct matvec::mult {
  template <class Matrix, class IterX, class IterY> inline
  matvec::mult(const col<Matrix>& A, IterX x, IterY y) {
    fast::inner_product(A.begin_columns(), cnt<N>(), x, y,
                        take1st(), add_op<M>(y));
  }
};

template <int M, class IterY>
class add_op {
public:
  add_op(IterY y_) : y(y_) {}
  template <class Col>
  IterY operator()(const Col& a, const T& x) {
    vecvec::add<M>(scale(a.begin(),x), y);
    return y;
  }
protected:
  const IterY y;
};
```

Fig. 5. BLAIS matrix-vector multiplication.

We then can use *iterators* and *2-dimensional iterators* to traverse the matrix. The *2-dimensional iterator* traverses the columns of the matrix. The expressions `A.begin_columns()` produces the *2-D column iterator*.

The use of `inner_product()` from the FAST Library as the outer loop may not be obvious at first. A matrix-vector multiplication $y = Ax$ can be thought of as the linear combination of the columns of `A` scaled by `x`. $y = A_0x_0 + A_1x_1 + \ldots + A_nx_n$. Abstractly, this linear combination can be thought of as an "inner product," where $y = \mathit{inner\_product}(A, x)$, and we think of A as a vector of columns.

3.3 Matrix-Matrix Operations

The BLAIS matrix-matrix operations are constructed using BLAIS matrix-vector routines. Note that although this approach is typically sub-optimal for arbitrary-length subroutines (i.e., ones that loop over matrices with run-time specified sizes), this approach does offer high performance in the fixed-size case because there are no actual loops. This is suitable for register-level blocking.

The BLAIS matrix-matrix multiplication operation can be created from matrix-vector products. The code is shown in Fig. 6. A recursive algorithm is used, and a call to the BLAIS matrix-vector multipy is made on each iteration.

```
// General case
template <int M, int N, int K> struct matmat::mult {
  template <class MatrixA, class Col2DIterB, class Col2DIterC> inline
  matmat::mult(const MatrixA& A, Col2DIterB Bcol, Col2DIterC Ccol) {
    matvec::mult<M,N>(A, (*Bcol).begin(), (*Ccol).begin());
    matmat::mult<M,N,K-1>(A, ++Bcol, ++Ccol);
  }
};
// K = 0 case
template <int M, int N> struct matmat::mult<M,N,0> {
  template <class MatrixA, class Col2DIterB, class Col2DIterC> inline
  matmat::mult(const MatrixA& A, Col2DIterB Bcol, Col2DIterC Ccol) {
    // do nothing
  }
};
```

Fig. 6. BLAIS matrix-matrix multiplication.

4 BLAIS in a General Matrix-Matrix Product

A typical use of the BLAIS kernels would be to construct linear algebra subroutines for arbitrarily sized objects. The fixed-size nature of the BLAIS routines make them well-suited to perform register-level blocking within a hierarchically blocked matrix-matrix multiplication. This is the approach used in the the Matrix Template Library [9], which is a high-performance numerical linear algebra library in C++ that uses generic programming techniques. Note that excessive code block is not a problem in MTL because the complete unrolling is only done for very small sized blocks.

```
for (jj = 0; jj < N; jj += BFN)
  for (ii = 0; ii < M; ii += BFM) {
    copy_block<double,BFM,BFN> c(C+ii*N+jj, BFM, BFN, N);
    for (kk = 0; kk < K; kk += BFK) {
      dense_contig2D_ref<double> a(A+ii*K+kk, BFM, BFK);
      dense_contig2D_ref<double> b(B + kk*N + jj, BFK, BFN);
      matmat::mult<BFM,BFN,BFK>(a, b, c);
    }
  }
}
```

Fig. 7. Use of BLAIS matrix-matrix operation within a general matrix-matrix product.

Fig. 7 shows the inner most set of blocking loops for a matrix-matrix multiply. The constants `BFM`, `BFN`, and `BFK` are blocking factors chosen so that `c` can fit into the registers.

4.1 A Configurable Recursive Matrix-Matrix Multiply

A high performance matrix-matrix multiply code is highly sensitive to the memory hierarchy of a machine, from the number of registers to the levels and sizes of cache. To obtain the highest performance, algorithmic blocking must be done at each level of the memory hierarchy. A natural way to formulate this is to write the matrix-matrix multiply in a recursive fashion, where each level of recursion performs blocking for a particular level of the memory hierarchy.

We take this approach in the MTL algorithm. The size and shapes of the blocks at each level are determined by the *blocking adapter*. Each adapter contains the information for the next level of blocking. In this way the recursive algorithm is determined by a recursive template data-structure (which is set up at compile time). The setup code for the matrix-matrix multiply is show in Fig. 8. This example blocks for just one level of cache, with 64 x 64 sized blocks. The small 4 x 2 blocks fit into registers. Note that these numbers would normally be constants that are set in a header file.

```
template <class MatA, class MatB, class MatC>
void matmat::mult(MatA& A, MatB& B, MatC& C) {
  MatA::RegisterBlock<4,1> A_L0;  MatA::Block<64,64> A_L1;
  MatB::RegisterBlock<1,2> B_L0;  MatB::Block<64,64> B_L1;
  MatC::CopyBlock<4,2> C_L0;      MatC::Block<64,64> C_L1;
  matmat::__mult(block(block(A, A_L0), A_L1),
                 block(block(B, B_L0), B_L1),
                 block(block(C, C_L0), C_L1));
}
```

Fig. 8. Setup for the recursive matrix-matrix product.

The recursive algorithm is listed in Fig. 9. There is a 2-D iterator (`A_k`, `B_k`, and `C_i`) for each matrix, as well as 1-D iterator (`A_ki`, `B_kj`, and `C_ij`). The matrices have been wrapped up with `blocked` matrix adapters, so that dereferencing the 1-D iterator results in a submatrix. The recursive call is then made on the submatrices `A_block`, `*B_kj`, and `*C_ij`.

The bottom most level of recursion is implemented with a separate function that makes the calls to the BLAIS matrix-matrix multiply, and "cleans up" the leftover edge pieces. Fig. 10 shows such a kernel without the edge cleanup code.

```
template <class MatA, class MatB, class MatC>
void matmat::__mult(MatA& A, MatB& B, MatC& C) {
  A_k = A.begin_columns(); B_k = B.begin_rows();
  while (not_at(A_k, A.end_columns())) {
    C_i = C.begin_rows(); A_ki = (*A_k).begin();
    while (not_at(C_i, C.end_rows())) {
      B_kj = (*B_k).begin(); C_ij = (*C_i).begin();
      MatA::Block A_block = *A_ki;
      while (not_at(B_kj, (*B_k).end())) {
        __mult(A_block, *B_kj, *C_ij);
        ++B_kj; ++C_ij;
      } ++C_i; ++A_ki;
    } ++A_k; ++B_k;
  }
}
```

Fig. 9. A recursive matrix-matrix product algorithm.

```
template <class MatA, class MatB, class MatC>
void matmat::__mult(MatA& A, MatB& B, MatC& C) {
  A_k = A.begin_rows(); C_k = C.begin_rows();
  while (not_at(A_k, A.end_rows())) {
    B_j = B.begin_columns(); C_kj = (*C_k).begin()
    while (not_at(B_j, B.end_columsn())) {
      B_ji = (*B_j).begin(); A_ki = (*A_k).begin();
      MatC::Block C_block = *C_kj;
      while (not_at(B_ji, (*B_j).end())) {
        blais_matmat::mult(*A_ki, *B_ji, C_block);
        ++B_ji; ++A_ki;
      }
      // cleanup of K left out
      ++B_j; ++C_kj;
    }
    // cleanup of N left out
    ++A_k; ++C_k;
  }
  // cleanup of M left out
}
```

Fig. 10. The matrix-matrix product L-1 kernel

4.2 Optimizing Cache Conflict Misses

Besides blocking, another important optimization that can be done with matrix-matrix multiply code. Typically utilization of the level-1 cache is much lower than one might expect due to cache conflict misses. This is especially apparent in direct mapped and low associativity caches. The way to minimize this problem is to copy the block of matrix A being accessed into a contiguous section of memory [5]. This allows the code to use blocking sizes closer to the size of the L-1 cache without inducing as many cache conflict misses.

It turns out that this optimization is straightforward to implement in our recursive matrix-matrix multiply. We already have block objects (submatrices `A_block`, `*B_j`, and `*C_j`) in Fig. 9. We modify the constructors for these objects to make a copy to a contiguous part of memory, and the destructors to copy the block back to the original matrix. This is especially nice since the optimization doesn't clutter the algorithm code, but instead the change is encapsulated in the `copy_block` matrix class.

4.3 Performance Experiments

Our claim is that the BLAIS Library provides a technique to achieve high-performance. We have performed experiments that have verified this claim. The compilers used were Kuck and Associates C++ [4] (for C++ to C translation), and the Sun Solaris C compiler with maximum available optimizations. The experiments were run on a Sun UltraSPARC 170E.

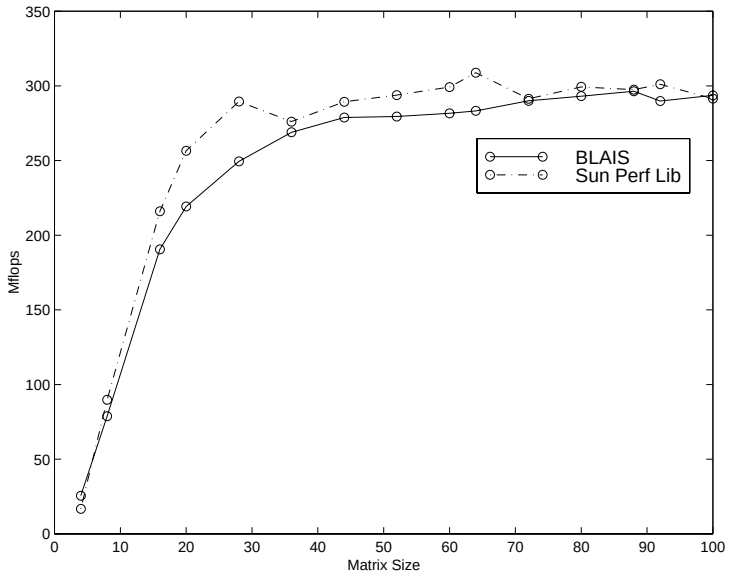

Fig. 11. Matrix-matrix multiply kernel.

Fig. 11 focuses on the performance of the BLAIS Library. The BLAIS routines are used for register blocking, so this experiment covers only small matrix sizes. Lager matrix sizes introduce more levels of blocking, which would cloud the picture.

The matrix-matrix multiply performance for the recursive MTL code is in Fig. 12. This experiment explores the full range of matrix sizes since the recursive algorithm blocks for each level of the memory hierarchy. We compare the MTL algorithm to the Sun Performance Library and the Netlib Fortran BLAS.

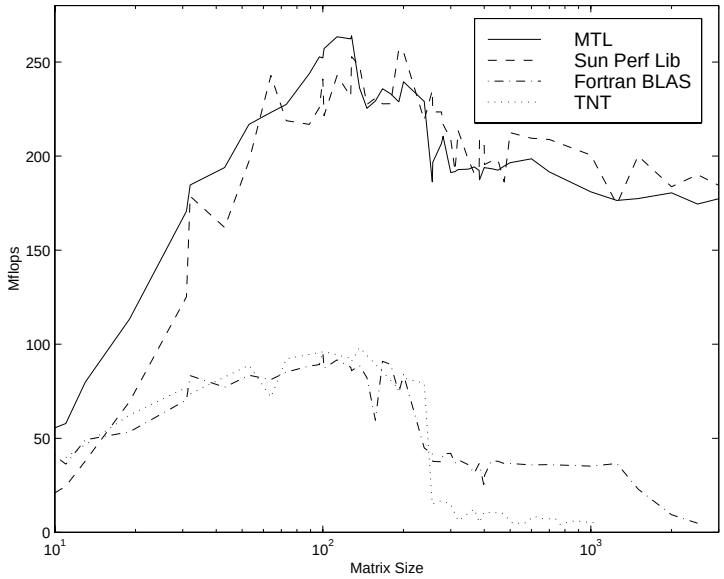

Fig. 12. General matrix-matrix multiply.

5 Availability

The BLAIS and FAST libraries are distributed as part of the Matrix Template Library, which can be downloaded from our web page at
`http://www.lsc.nd.edu/research/mtl/`

Acknowledgments

This work was supported by NSF grants ASC94-22380 and CCR95-02710. The authors would like to express their appreciation to Tony Skjellum and Puri Bangalore for numerous helpful discussions.

References

1. J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.-W. Chin. 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.
2. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.
3. J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of basic linear algebra subprograms: Model implementations and test programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.
4. Kuck and Associates. *Kuck and Associates C++ User's Guide*.
5. Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performacen and optimizations of blocked algorithms. In *ASPLOS IV*, April 1991.
6. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.
7. Meng Lee and Alexander Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.
8. Jack J. Dongarra R. Clint Whaley. Automatically tuned linear algebra software (atlas). Technical report, University of Tennessee and Oak Ridge National Laboratory, 1997.
9. Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A unifying framework for numerical linear algebra. In *Parallel Object Oriented Scientific Computing*. ECOOP, 1998.
10. Todd Veldhuizen. Using c++ template metaprograms. *C++ Report*, May 1995.