The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra [*]

Jeremy G. Siek   Andrew Lumsdaine

Laboratory for Scientific Computing
Department of Computer Science and Engineering
University of Notre Dame

Abstract. We present a unified approach for expressing high performance numerical linear algebra routines for large classes of dense and sparse matrices. As with the Standard Template Library [10], we explicitly separate algorithms from data structures through the use of generic programming techniques. We conclude that such an approach does not hinder high performance. On the contrary, writing portable high performance codes is actually enabled with such an approach because the performance critical code sections can be isolated from the algorithms and the data structures. We also tackle the performance portability problem for particular architecture dependent algorithms such as matrix-matrix multiply. Recently, code generation systems (PHiPAC [3] and ATLAS [15]) have been created to customize the algorithms according to architecture. A more elegant approach is to use *template metaprograms* [18] to allow for variation. In this paper we introduce the Basic Linear Algebra Instruction Set (BLAIS), a collection of high performance kernels for basic linear algebra.

1 Introduction

The traditional approach to writing basic linear algebra routines is a combinatorial affair. There are typically four precision types that need to be handled (single and double precision real, single and double precision complex), several dense storage types (general, banded, packed), a multitude of sparse storage types (13 in the Sparse BLAS Standard Proposal [1]), as well as row and column orientations for each matrix type. To provide a full implementation one might need to code literally hundreds of versions of the same routine! It is no wonder the NIST implementation of the Sparse BLAS contains over 10,000 routines and an automatic code generation system [16].

To make matters worse, the performance of codes such as matrix-matrix multiply is highly sensitive to the memory hierarchy characteristics, so writing portable high-performance codes is even more difficult. It is typically necessary to use a code generation system on top of C or Fortran in order to get the flexibility needed for register blocking according to computer architecture.

In this paper we apply the fundamental generic programming approaches used by the Standard Template Library (STL) to the domain of numerical linear algebra. The resulting library, which we call the *Matrix Template Library* (MTL) provides comprehensive functionality with a small number of of fundamental algorithms, while at the

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

same time achieving high performance. We also explore the use of *template metaprograms* in the construction of the BLAIS kernels, which provide an elegant solution to portable high performance for matrix-matrix multiply and other blocked codes.

The Matrix Template Library [11] is in its second generation, and has been completely rewritten using generic programming techniques.

2 Generic Programming

The principal idea behind the STL is that many algorithms can be abstracted away from the particular data structures on which they operate. Algorithms typically need the abstract functionality of being able to *traverse* through a data structure and *access* its elements. If data structures provide a standard interface for traversal and access, generic algorithms can be mixed and matched with data structures (called *containers* in STL). This interface is realized through the *iterator* (sometimes called a generalized pointer).

Abstractly, linear algebra operations also consist of traversing through vectors and matrices. Vector operations fit neatly into the generic programming approach. The STL already defines several generic algorithms for vectors, such as `inner_product()`. Extending these generic algorithms to encompass the rest of the common Level-1 BLAS [9] is a trivial matter.

```
template <class Row2DIter, class IterX, class IterY> void
matvec::mult(Row2DIter i, Row2DIter iend, IterX x, IterY y) {
  typename Row2DIter::value_type::const_iterator j;
  while (not_at(i, iend)) {
    j = (*i).begin();
    typename IterY::value_type tmp(0);
    while (not_at(j, (*i).end())) {
      tmp += *j * x[j.index()];
      ++j;
    }
    y[i.index()] = tmp;
    ++i;
  }
}
```

Fig. 1. Simplified example of a generic matrix-vector product.

Matrix operations are slightly more complex, since the elements are arranged in a 2-dimensional format. The MTL processes matrices as if they are *containers of containers* (note that the matrix implementations are typically not actual containers of containers). The matrix algorithms are coded in terms of *iterators* and *two-dimensional iterators*. A `Row2DIter` can traverse the rows of a matrix, and produces a row vector when dereferenced. The iterator for the row vector can then be used to access the individual matrix elements. The example in Fig. 1 shows how one can write a generic matrix-vector product.

Function Name	**Operation**	**Function Name**	**Operation**
Vector Algorithms		Vector Vector	
`set(x,alpha)`	$x_i \leftarrow \alpha$	`copy(x,y)`	$y \leftarrow x$
`scale(x,alpha)`	$x \leftarrow \alpha x$	`swap(x,y)`	$y \leftrightarrow x$
`s = sum(x)`	$s \leftarrow \sum_i x_i$	`ele_mult(x,y,z)`	$z \leftarrow y \otimes x$
`s = one_norm(x)`	$s \leftarrow \sum_i \mid x_i \mid$	`ele_div(x,y,z)`	$z \leftarrow y \oslash x$
`s = two_norm(x)`	$s \leftarrow (\sum_i x_i^2)^{\frac{1}{2}}$	`add(x,y)`	$y \leftarrow x + y$
`s = inf_norm(x)`	$s \leftarrow$ max $\mid x_i \mid$	`s = dot(x,y)`	$s \leftarrow x^T \cdot y$
`i = find_max_abs(x)`	$i \leftarrow$ index of max $\mid x_i \mid$	`s = dot_conj(x,y)`	$s \leftarrow x^T \cdot \bar{y}$
`s = max(x)`	$s \leftarrow$max(x_i)		
`s = min(x)`	$s \leftarrow$min(x_i)		
Matrix Algorithms		Matrix Vector	
`set(A, alpha)`	$A \leftarrow \alpha$	`mult(A,x,y)`	$y \leftarrow A \times x$
`scale(A,alpha)`	$A \leftarrow \alpha A$	`mult(A,x,y,z)`	$z \leftarrow A \times x + y$
`set_diag(A,alpha)`	$A_{ii} \leftarrow \alpha$	`tri_solve(T,x,y)`	$y \leftarrow T^{-1} \times x$
`s = one_norm(A)`	$s \leftarrow max_i(\sum_j \mid a_{ij} \mid)$	`rank_one(x,A)`	$A \leftarrow x \times y^T + A$
`s = inf_norm(A)`	$s \leftarrow max_j(\sum_i \mid a_{ij} \mid)$	`rank_two(x,y,A)`	$A \leftarrow x \times y^T +$ $y \times x^T + A$
`transpose(A)`	$A \leftarrow A^T$		
Matrix Matrix			
`copy(A,B)`	$B \leftarrow A$	`swap(A,B)`	$B \leftrightarrow A$
`add(A,C)`	$C \leftarrow A + C$	`ele_mult(A,B,C)`	$C \leftarrow B \otimes A$
`mult(A,B,C)`	$C \leftarrow A \times B$	`mult(A,B,C,E)`	$E \leftarrow A \times B + C$
`tri_solve(T,B,C)`	$C \leftarrow T^{-1} \times B$		

Table 1. MTL linear algebra operations.

3 MTL Algorithms

Table 1 lists the principal algorithms covered by MTL. This list seems sparse, but a large number of functions are indeed provided through the combination of the above algorithms with the `strided()`, `scaled()`, and `trans()` adapter functions. Fig. 2 shows how this is done with a matrix-vector multiply and with a scaled vector assignment.

The unique feature of the Matrix Template Library is that, for the most part, each of the algorithms is implemented with just one template function. Just one algorithm is used whether the matrix is sparse, dense, banded, single precision, double, complex, etc. From a software maintenance standpoint, the reuse of code gives MTL a significant advantage over the BLAS [4, 5] or even other object-oriented libraries like TNT [14] (which has different algorithms for different matrix formats).

The generic algorithm code reuse results in the MTL having 10 times fewer lines of code than the netlib Fortran BLAS while providing greater functionality and achieving

```
//   y <- A * alpha x
matvec::mult(trans(A), scaled(x, alpha), strided(y,incy));
//   y <- alpha x
vecvec::copy(scaled(x, alpha), y);
```

Fig. 2. Transpose, Scaled, and Strided Adapters

generally better performance, especially for level 2 and 3 operations. The MTL has 8,284 lines of code for the algorithms and 6,900 lines of code for dense containers, for a total of 15,184 lines of code. The Fortran BLAS total 154,495 lines of code, an order of magnitude more.

4 MTL Components

The Matrix Template Library defines a set of data structures and other components for representing linear algebra objects. An MTL matrix is constructed with layers of components. Each layer is a collection of classes that are templated on the lower layer. The bottom most layer consists of the numerical types (`float`, `double`, etc). The next layers consist of 1-D containers followed by 2-D containers. The 2-D containers are wrapped up with an *orientation*, which in turn is wrapped with a *shape*. A complete MTL matrix type typically consists of a templated expression in the form `shape < orientation < twod < oned < num_type > > > >`. For example, an upper triangular matrix would be defined as `triangle< column< dense2D< double > >, upper>`. Some 2-D containers also subsume the 1-D type, such as the contiguous `dense2D` container.

Matrix Orientation The `row` and `column` adapters map the *major* and *minor* aspects of a matrix to the corresponding *row* or *column*. This technique allows the same code for data structures to provide both row and column orientations of the matrix. 2-D containers must be wrapped up with one of these adapters to be used in the MTL algorithms.

Matrix Shape Matrices can be categorized into several shapes: general, upper triangular, lower triangular, symmetric, Hermitian, etc. The traditional approach to handling the algorithmic differences due to shape is to have a separate function for each type. For instance, in the BLAS we have a `_GEMV`, `_SYMV`, `_TRMV`, etc. The MTL instead uses different data structures for each shape, with the `banded`, `triangle`, `symmetric`, and `hermitian` matrix adapters. It is the responsibility of these adapters to make sure that they work with all of the MTL generic algorithms. The MTL philosophy is to use *smarter* data structures to allow for fewer and simpler algorithms.

5 The High Performance Layer

We have presented many levels of abstraction, and a set of unified algorithms for a variety of matrices, but this matters little if high performance can not be achieved. Template based programming coupled with modern compilers such as Kuck and Associates (KAI) C++ [7] provide several mechanisms for high-performance.

Static Polymorphism The template facilities in C++ allow functions to be selected at compile-time based on data type. This provides a mechanism for abstraction which preserves high performance. Dynamic (run-time) dispatch is avoided, and the template functions can be inlined just as regular functions. This ensures that the numerous small function calls in the MTL (such as iterator increment operators) introduce no extra overhead.

Lightweight Object Optimization The generic programming style introduces a large number of small objects into the code. This incurs a performance penalty because the presence of a structure can interfere with other optimizations, including the mapping of the individual data items to registers. This problem is solved with small object optimization, also know as scalar replacement of aggregates [12], which is performed by the KAI C++ compiler.

Automatic Unrolling Modern compilers can do a great job of unrolling loops and scheduling instructions, but typically only for specific (recognizable) cases. There are many ways, especially in C and C++ to interfere with the optimization process. The abstractions of the MTL are designed to result in code that is easy for the compiler to optimize. Furthermore, the *iterator* abstraction makes inter-compiler portability possible, since it encapsulates how looping is performed.

Algorithmic Blocking 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 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 [3], ATLAS [15]) have resorted to going outside the language, i.e., to code generation systems, in order to gain this kind of flexibility. In the following sections 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 at a high level.

5.1 The Basic Linear Algebra Instruction Set (BLAIS)

The BLAIS specification contains *fixed size* algorithms with functionality equivalent to that of the Level-1, Level-2, and Level-3 BLAS [4, 5, 9]. 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 STL. The thin BLAIS routines merely map the 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 simple and elegant fashion. Note that the intended use of the BLAIS routines is to carry out the register blocking within a larger algorithm. This means the BLAIS routines handle only small matrices, and therefore avoid the problem of excessive code bloat.

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 handle the register level blocking in a matrix-matrix product. Finally, experimental results show that the performance obtained by our approach can equal and even exceed that of vendor-tuned libraries.

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*. The only difference is that the loop-end iterator is replaced by a *count template* object. The example shown in Fig. 3 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.

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

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. 4, 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 [19].

```
// 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 InItr1,class InItr2,class OutItr,class BinOp>
inline OutItr
fast::transform (InItr1 first1, cnt<0>, InItr2 first2,
                 OutItr result, BinOp binary_op) {
  return result; }
```

Fig. 4. Definition of FAST `transform()`.

BLAIS Vector-Vector Operations Fig. 5 gives the implementation for the BLAIS vector `add()` routine, and shows an example of its use. The FAST `transform()` algorithm is used to carry out the vector-vector addition as it was in the example above.

The comments on the right show the resulting code after the call to `add()` is inlined. The `scl()` function used above demonstrates the purpose of the `scale_iterator`. The `scale_iterator` multiplies the value from `x` by `a` when the iterator is dereferenced within the `add()` routine. This adds no extra time or space overhead due to inlining and lightweight object optimizations. The `scl(x, a)` call automatically creates the proper `scale_iterator` out of `x` and `a`.

BLAIS Matrix-Vector Operations The BLAIS matrix-vector multiply implementation is depicted in Fig. 6. The algorithm simply carries out the vector add operation for the columns of the matrix. Again a fixed depth recursize algorithm is used, which becomes inlined by the compiler.

BLAIS Matrix-Matrix Operations The BLAIS matrix-matrix multiply is implemented using the BLAIS matrix-vector operation. The code looks very similar to the matrix vector multiply, except that there are three integer template arguments (M, N, and K), and the inner “loop” contains a call to `matvec::mult()` instead of `vecvec::add()`.

```
// Definition
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>());
}};
// Example use
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. 5. Definition an Use of BLAIS `add()`.

5.2 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. 7. 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.

The recursive algorithm is listed in Fig. 8. The bottom most level of recursion is implemented with a separate function that uses the BLAIS matrix-matrix multiply, and "cleans up" the leftover edge pieces.

5.3 Optimizing Cache Conflict Misses

Besides blocking, another important optimization that can be done with matrix-matrix multiply code is to perform block copies. 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 current block of matrix A into a contiguous section of memory [8]. 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`,

```
// General Case
template <int M, int N>
struct mult {
  template <class AColIter, class IterX, class IterY> inline
  mult(AColIter A_2Diter, IterX x, IterY y) {
    vecvec::add<M>(scl((*A_2Diter).begin(), *x), y);
    mult<M, N-1>(++A_2Diter, ++x, y);
  }
};
// N = 0 Case
template <int M>
struct mult<M, 0> {
  template <class AColIter, class IterX, class IterY> inline
  mult(AColIter A_2Diter, IterX x, IterY y) {
    // do nothing
  }
};
```

Fig. 6. BLAIS matrix-vector multiplication.

```
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_LO), A_L1),
                 block(block(B, B_L0), B_L1),
                 block(block(C, C_L0), C_L1));
}
```

Fig. 7. Setup for the recursive matrix-matrix product.

and `*C_j`) in Fig. 8. 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 does not clutter the algorithm code, but instead the change is encapsulated in the `copy_block` matrix class.

6 Performance Experiments

The compilers used were Kuck and Associates C++ [7] (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. 9 shows the sparse matrix vector performance for MTL, SPARSKIT [17] (Fortran), NIST [16](C), and TNT (C++). The sparse matrices used are from the MatrixMarket [13] collection. The matrix-matrix multiply

```
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. 8. A recursive matrix-matrix product algorithm.

performance for MTL, the Sun Performance Library, TNT, and the Netlib Fortran BLAS is shown in Fig. 10.

7 Supplemental Libraries

The Matrix Template Library provides a nice foundation for constructing portable high performance libraries. We have created two such libraries, the ITL and the BLAS. The Iterative Template Library (ITL) is a collection of sophisticated iterative solvers similar to the Iterative Methods Library (IML)[6]. It calls the MTL for its basic linear algebra operations. We have implemented the standard BLAS routines using the MTL data structures and algorithms. Our initial tests show that the performance is comparable (within $\pm 5\%$) to the Fortran BLAS and vendor-tuned BLAS.

Additionally, we have provided an MTL interface to LAPACK [2], so that users of MTL have a convenient way to access the LAPACK functionality.

8 Conclusion

Recent attempts to create portable high performance linear algebra routines have relied upon specialized code generation scripts in order in provide enough flexibility in C and Fortran codes. In this paper we have shown that C++ has enough expressiveness to allow codes to be reconfigured for particular architectures by merely changing a few constants. In addition, we have found that advanced C++ compilers can still aggressively optimize in the presence of the powerful MTL abstractions, producing code that matches or exceeds the performance of hand coded C and vendor-tuned libraries.

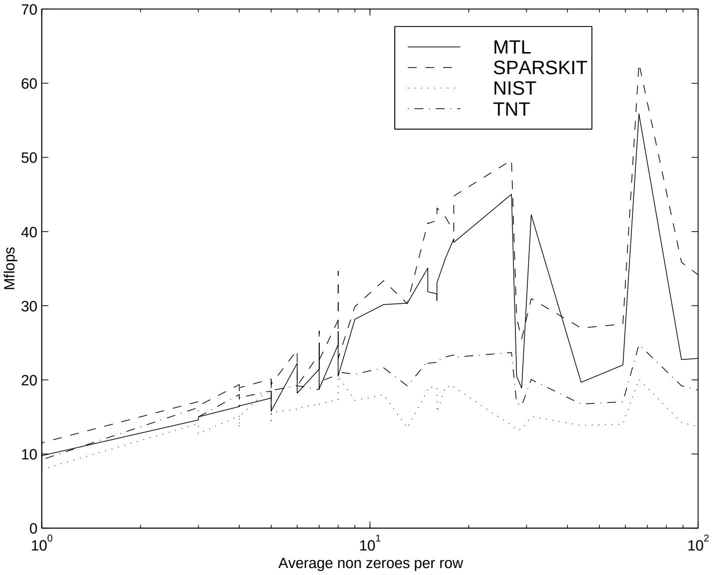

Fig. 9. Sparse matrix-vector multiply.

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. BLAS standard draft chapter 3: Sparse BLAS. Technical report, Basic Linear Algebra Subprograms Technical Forum, December 1997.
2. E. Anderson, Z. Bai, C. Bischoff, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra package for high-performance computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.
3. 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.
4. 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.
5. 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.
6. Roldan Pozo Jack Dongarra, Andrew Lumsdaine and Karin A. Remington. *Iterative Methods Library Reference Guide*, v. 1.2 edition, 1997.

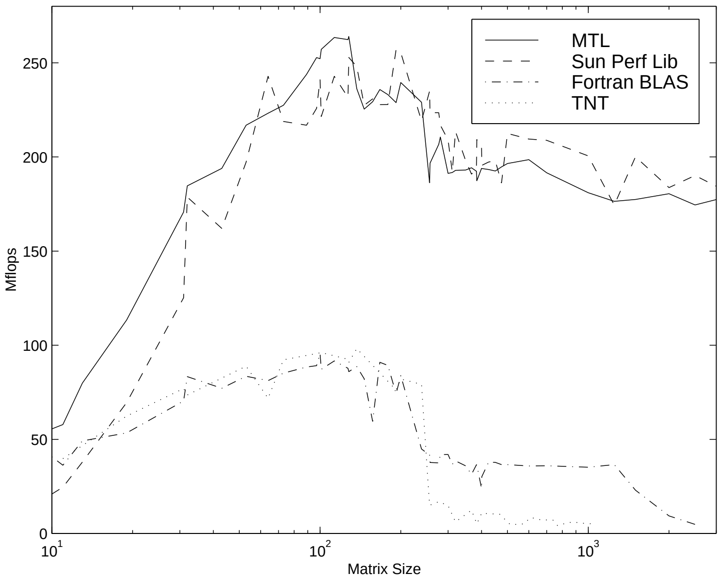

Fig. 10. General matrix-matrix multiply.

7. Kuck and Associates. *Kuck and Associates C++ User's Guide.*
8. Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS IV*, April 1991.
9. 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.
10. Meng Lee and Alexander Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.
11. Brian C. McCandless and Andrew Lumsdaine. The role of abstraction in high-performance computing. In *Scientific Computing in Object-Oriented Parallel Environments*. ISCOPE, December 1997.
12. Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
13. NIST. MatrixMarket. http://gams.nist.gov/MatrixMarket/.
14. Roldan Pozo. *Template Numerical Toolkit (TNT) for Linear Algebra.* National Insitute of Standards and Technology.
15. Jack J. Dongarra R. Clint Whaley. Automatically tuned linear algebra software (ATLAS). Technical report, University of Tennessee and Oak Ridge National Laboratory, 1997.
16. Karen A. Remington and Roldan Pozo. *NIST Sparse BLAS User's Guide.* National Institute of Standards and Technology.
17. Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Technical report, NASA Ames Research Center, 1990.
18. Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, May 1995.
19. Todd Veldhuizen and M. Ed Jernigan. Will C++ be faster than Fortran. In *Scientific Computing in Object-Oriented Parallel Environments.* ISCOPE, December 1997.