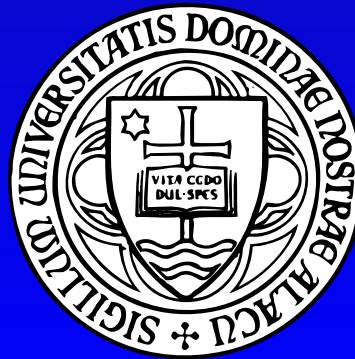


Generic Programming for High Performance Numerical Linear Algebra

Andrew Lumsdaine and Jeremy Siek

Department of Computer Science and Engineering

University of Notre Dame



Overview

- Motivation
- Generic algorithms for linear algebra
- Design of MTL and BLAIS
- Performance results
- Conclusions

Motivation

- Scientific computing is one of the most important and successful genres of computing
- Exceedingly complex, therefore sound software engineering needed even more than in other genres
- Scientific software is mired in Fortran 77
- Our goal: Provide a complete, modern, high performance library for numerical linear algebra
- Our hammer: **Genericity**

Linear Algebra

The domain abstractions:

- Vector space V over a field F
- Scalar arithmetic operations for F
- Vector space operations for V
- Linear transformations : $V \rightarrow V$

Linear Algebra

- $\alpha, \beta \in F, x, y \in V$

Scalar multiply

$$\alpha * x \in V$$

Linearity

$$(\alpha * x + \beta * y) \in V$$

Linear transformation

$$A : V \rightarrow V$$

Additional Structure

- $\alpha, \beta \in F, x, y \in V$

Banach space $\|x\|$

Hilbert space $\langle x, y \rangle$

Dual space $A^* : V^* \rightarrow V^*$

Generic Algorithms for Linear Algebra

- A complete set of linear algebra functionality:

Generic algorithm	Abstract operation
<code>scale()</code>	$\alpha * x \in V$
<code>add()</code>	$(\alpha * x + \beta * y) \in V$
<code>mult()</code>	$A : V \rightarrow V$
<code>norm()</code>	$\ x\ $
<code>dot()</code>	$\langle x, y \rangle$
<code>transpose()</code>	$A^* : V^* \rightarrow V^*$

Conjugate Gradient Algorithm

for $i = 1, 2, \dots$

solve $Mz^{(i-1)} = r^{(i-1)}$

$$\rho_{i-1} = r^{(i-1)T} z^{(i-1)}$$

if $i = 1$

$$p^{(1)} = z^{(0)}$$

else

$$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$$

$$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$$

$$q^{(i)} = Ap^{(i)}, \alpha_i = \rho_{i-1} / p^{(i)T} q^{(i)}$$

$$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}, r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$$

check convergence

end

ITL Implementation of CG

```
while (! iter.finished(r)) {
    M.solve(r, z);
    rho = dot_conj(r, z);
    if (iter.first()) copy(z, p);
    else { beta = rho / rho_1;
           add(z, scaled(p, beta), p); }
    mult(A, p, q);
    alpha = rho / dot_conj(p, q);
    add(x, scaled(p, alpha), x);
    add(r, scaled(q, -alpha), r);
    rho_1 = rho;
    ++iter;
}
```

Genericity and Abstraction

- Genericity and abstraction are dual concepts
- Ideally, an abstract operation should be represented as a single, generic algorithm
- E.g., an `add()` algorithm should implement $(\alpha * x + \beta * y) \in V$ for any concrete representation of V
- In the real world, we *classify* representations and provide generic algorithms for each class (*not* each `class`)

Genericity and C++

- Generic programming can be accomplished in C++ with the template system

```
template <class InIter, class T>
T accumulate(InIter first, InIter last, T init)
{
    while (first != last)
        init = init + *first++;
    return init;
}
```

Matrix Template Library

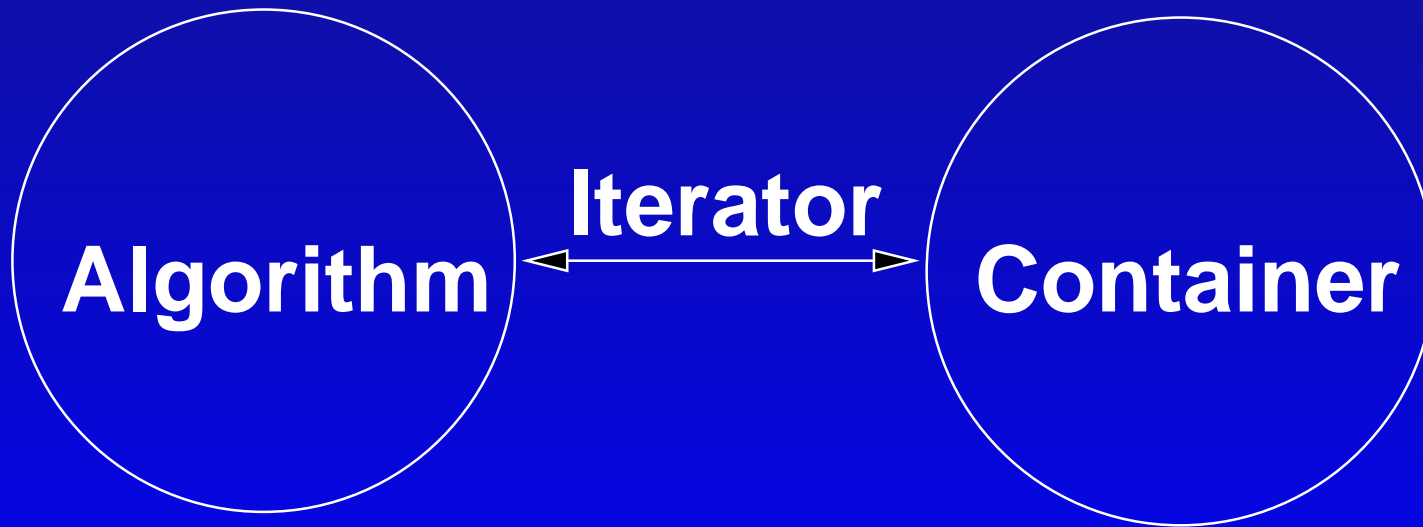
- A complete framework is needed, operating in conjunction with the generic algorithms
- The Matrix Template Library (MTL) includes
 - { Generic algorithms
 - { Containers
 - { Iterators
 - { Adaptors
 - { Function objects

The MTL Generic Algorithms

- As with the STL, use *iterators* to traverse through a container
- A matrix representation can be abstractly thought of as a *container of containers*
- Use *iterators* and *2-dimensional iterators* to traverse the matrix
- A large class of matrix types can be implemented with this interface

Iterators for Linear Algebra

- Interface between generic algorithms and containers



Index-free Algorithms

- Iterate from `begin()` to `end()` of a vector.
- Iterate from `begin_rows()` to `end_rows()` (or `end_columns()`) of a 2-D container.
- This side-steps traditional annoyances such as the difference between Fortran (from 1) and C (from 0) indexing.

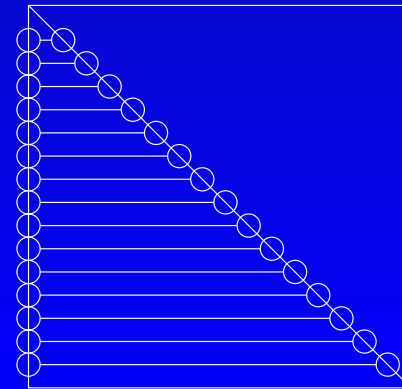
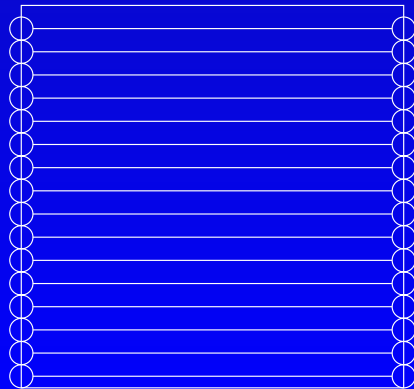
Unifying Sparse and Dense

- Iterators hides difference in traversal
- `index()` method hides difference in indexing
- An example from a matrix-vector multiply.

```
for(j = i->begin(); j != i->end(); ++j)
    tmp += *j * x[j.index()];
```

Same Algorithm with Shaped Matrices

- banded, triangle, symmetric, hermitian
- Algorithms process vectors from `begin()` to `end()`.
Therefore container implementations change, algorithms do not.



A Generic Matrix-Vector Multiply

```
template <class Matrix, class VecX, class VecY>
void mult(Matrix A, VecX x, VecY y) {
    typename Matrix::row_2Diterator i;
    typename Matrix::RowVector::iterator j;
    for (i = A.begin_rows();
         i != A.end_rows(); ++i) {
        typename Matrix::value_type
            tmp = y[i.index()];
        for(j = i->begin(); j != i->end(); ++j)
            tmp += *j * x[j.index()];
        y[i.index()] = tmp;
    }
}
```

Transpose, Scaling, and Striding Permutations

- Avoid repetitious algorithm variations as in the BLAS
- Use matrix, vector, and iterator adaptors instead
- An adaptor wraps up an object and modifies its behavior

```
// y <- A' * alpha x
```

```
mult(trans(A), scaled(x, alpha),  
      strided(y, incy));
```

MTL Iterators

- MTL iterators include **sparse**, **dense**, **strided**, **scaled**, and **block**
- Iterator adaptors modify the behavior of a base iterator

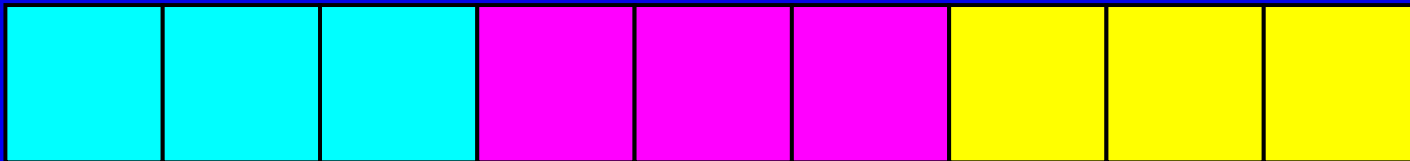
```
template <class Iter>
class scaled_iterator {
    scale_iterator(Iter i, T a)
        : iter(i), alpha(a) { }
    T operator*() { return *iter * alpha; }
    self& operator++() { ++iter; return *this; }
    Iter iter; T alpha;
};
```

Containers for Linear Algebra

- Concrete representations of members of V (vectors) and linear transformations (matrices)
- Vectors are relatively straightforward (one dimensional container of type F , similar to `STL vector`)
- Matrices are more involved because two-dimensional object must be implemented with a one-dimensional memory space

Containers for Linear Algebra: Properties

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Leftrightarrow A(i, j)$$



Containers for Linear Algebra:

Properties

Two-dimensional storage: e.g., dense contiguous, compressed sparse

Underlying one-dimensional storage: e.g., `list`

Basic (elemental) type: e.g., `float`, `complex<double>`

Orientation: e.g., row, column, diagonal

Shape: e.g., symmetric, triangle, banded

MTL Containers for Linear Algebra

- MTL matrices are built by template composition of
 - { Basic numeric type (precision) (x6)
 - { One-dimensional container (x5)
 - { Two-dimensional container (x2)
 - { Orientation (x3)
 - { Shape and packing (x8)
- Approx 1440 matrix types implemented in 16 classes

```
triangle<row<array2D<dense1D<double>>>, lower>
```

Role for each MTL template layer

- Numeric Type: numeric representation and arithmetic
- Storage: determine elt location in memory
- Shape: transform (major,minor) to a different “shape”
- Orientation: map (major,minor) to (row,column)

```
triangle<row<array2D<dense1D<double>>>,lower>
```

Demonstration of the roles

- Example implementations of the `operator()(i, j)`.
- Storage layer: map 2-D coords to a linear memory

```
// normal dense matrix storage
reference operator()(int major_index,
                    int minor_index) {
    return *(major_index * ld + minor_index);
}
```

Roles: more storage formats

```
// banded matrix storage
reference operator()(int major_index,
                    int minor_index) {
    return *(i * ndiag
            + max(0, sub(bandwidth) - i) + j);
}

// array of array storage
reference operator()(int major_index,
                    int minor_index) {
    return array[major_index][minor_index];
}
```

Roles: Shape Adaptor Level

```
// banded shape adaptor
reference operator()(int major_index,
                    int minor_index) {
    return twod(major_index, minor_index
                - max(major_index - sub(bandwidth), 0));
}
```

Roles: Shape Adaptor Level

```
// diagonal shape adaptor
reference operator()(int major_index,
                    int minor_index) {
    size_type diag = major_index -
                    minor_index + super(bandwidth);
    if (major_index >= minor_index)
        return twod(diag, minor_index);
    else
        return twod(diag, major_index);
}
```

Roles: Orientation Adaptor Level

```
// row-major adaptor
reference operator()(int i, int j) {
    return shape(i,j);
}

// column-major adaptor
reference operator()(int i, int j) {
    return shape(j,i);
}
```

The MTL Basic Numeric Types

MTL can support an unlimited number of basic numeric types. The following types have been tested in conjunction with MTL.

Basic C++ types: bool, int, float, double

Extended Precision: doubledouble

Complex: complex<float>, complex<double>, complex<doubledouble>

Interval: interval<float>, etc.

The MTL 1-D Containers

- `dense1D` implemented with STL's `vector` class
- `sparse1D` adaptor for containers of index-value pairs
- `compressed1D` separate index and value arrays
- `scaled1D` and `strided1D` adaptor classes

```
triangle<row<array2D<dense1D<double>>>, lower>
```

The MTL 2-D Containers

- `array2D` composes 1-D containers into a matrix
- `dense2D` contiguous dense matrix
- `compressed2D` contiguous sparse matrix
- `scaled2D` adaptor class

```
triangle<row<array2D<dense1D<double>>>, lower>
```

The MTL Orientation Adaptors

Use a single 2-D container implementation for both row and column major matrix formats.

- Row Orientation

- { Maps row to *major*, column to *minor*

- Column Orientation

- { Maps column to *major*, row to *minor*

- All 2-D methods and typedefs are mapped.

The MTL Shape Adaptors

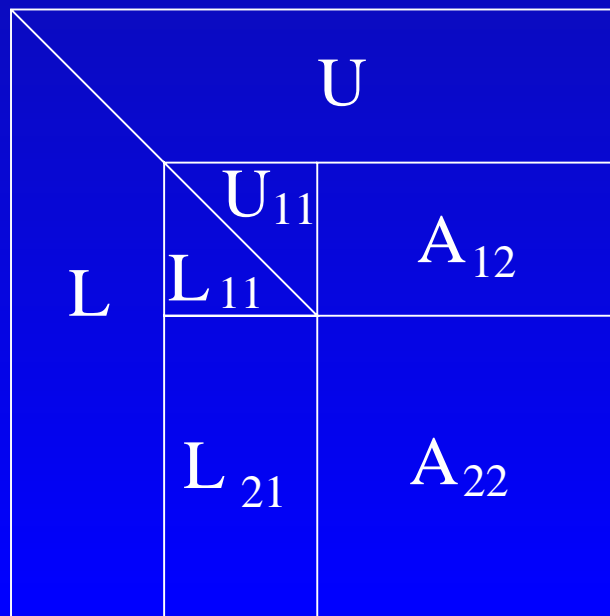
- banded
- triangle
- symmetric
- hermitian

Each in packed and unpacked variations.

```
triangle<row<array2D<dense1D<double>>>,lower>
```

Example Use: Block LU Algorithm

- Perform point-wise LU to get L_{11} , U_{11} , and L_{21} .
- Do a triangular solve to get A_{12} .
- Do matrix product $A_{22} \leftarrow L_{21} \times A_{12}$.



The Block LU Implementation

```
void block_lu(Matrix& A, Pvector& ipvt) {
    Pvector pivots(BF);
    for (int j = 0; j < min(M, N); j += BF) {
        int jb = min(min(M, N) - j, BF);
        // set up the submatrices A0, A1, A2
        //      L11, A12, A21, A22 ...
        lu_factorize(A1, pivots);
        multi_swap(A0, pivots);
        if (j + jb < M) {
            multi_swap(A2, pivots);
            tri_solve(L11, A12, left_side());
            if (j + jb < M)
                mult(scaled(A21, -1), A12, A22);
        }
    }
}
```

Example Use:

The Iterative Template Library (ITL)

- Iterative methods:
CG, CGS, BiCG, BiCGStab, GMRES, QMR, TFQMR,
Chebyshev, Richardson
- Preconditioners:
SSOR, incomplete Cholesky, incomplete LU, ILU with
thresholding
- Generic interface and implementation
built on top of the MTL

The ITL Generic Components

Perconditioner An object with a `solve(x, z)` method.

Iteration Convergence test and iteration count.

Vector STL-like container with iterator interface.

Matrix Either a MTL style matrix or a *multiplier* object for matrix-free methods.

The ITL Function Interface

The interface for the quasi-minimal residual (QMR) method, which uses a split preconditioner, hence M1 and M2.

```
template <class Matrix, class Vector1,  
          class Vector2, class Precond1,  
          class Precond2, class Iteration>  
int qmr(const Matrix& A, Vector1& x,  
        const Vector2& b, const Precond1& M1,  
        const Precond2& M2, Iteration& iter);
```

Usage of ITL QMR Method

```
typedef row< compressed2D<double> > matrix;  
int max_iter = 50;  
matrix A(5, 5);  
dense1D<double> x(A.nrows(), 0.0);  
dense1D<double> b(A.ncols(), 1.0);  
// fill A ...  
SSOR<matrix> precondition(A);  
basic_iteration<double> iter(b,max_iter,1e-6);  
qmr(A,x,b,precond.left(),precond.right(),iter);
```

Conjugate Gradient Algorithm

for $i = 1, 2, \dots$

solve $Mz^{(i-1)} = r^{(i-1)}$

$$\rho_{i-1} = r^{(i-1)T} z^{(i-1)}$$

if $i = 1$

$$p^{(1)} = z^{(0)}$$

else

$$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$$

$$p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$$

$$q^{(i)} = Ap^{(i)}, \alpha_i = \rho_{i-1} / p^{(i)T} q^{(i)}$$

$$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}, r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$$

check convergence

end

ITL Implementation of CG

```
while (! iter.finished(r)) {  
    M.solve(r, z);  
    rho = dot_conj(r, z);  
    if (iter.first()) copy(z, p);  
    else { beta = rho / rho_1;  
           add(z, scaled(p, beta), p); }  
    mult(A, p, q);  
    alpha = rho / dot_conj(p, q);  
    add(x, scaled(p, alpha), x);  
    add(r, scaled(q, -alpha), r);  
    rho_1 = rho;  
    ++iter;  
}
```

The ITL Advantages

- Mix and match components
- Flexible and extensible
- Easy to implement, maintain, and modify
- Performance issues separated from algorithms
- Matrix storage types separated from algorithms

Fortran-like performance with C++

That is, high performance amidst the use of abstractions.

Static Polymorphism: Templates allow for data-type based function selection at compile time. This enables **inlining**.

Lightweight Object Optimization: The compiler reduces structures to their parts to enable register allocation. This means iterators can be high performance.

Follow Coding Guidelines: Enable the C++ compiler to make the above optimizations. Double check the intermediate C code.

Performance Digression

- It is easy to get high levels of performance if you define “high” in a suitable (but vague) fashion
 - { “Equivalent to C / Fortran”
 - { “Equivalent to hand-written”
 - { “Within a factor of 2 of C / Fortran”
- Vendor-tuned are typically (but not always) the best
- MTL performance goal is to offer best possible performance

Peak Performance

- Tuning process is fairly well understood
- Make use of architecture features that are provided for high performance
 - { Cache (improve temporal and spatial locality)
 - { Instruction pipelining
- These processes can be parameterized
- Create abstractions to handle the optimizations

Meta-Programming Optimizations

Important optimizations to perform in the C/C++ code:

- Loop unrolling
- Register-level blocking

The blocking sizes are machine dependent, so the optimization scheme must be flexible and parameterized, which is impossible to do in C or Fortran.

The Language Problem

It is impossible to express variable degrees of unrolling and blocking in C and Fortran

```
// unroll by two
```

```
y[0] += a * x[0];
```

```
y[1] += a * x[1];
```

```
// unroll by three
```

```
y[0] += a * x[0];
```

```
y[1] += a * x[1];
```

```
y[2] += a * x[2];
```

Previous Solutions: PHiPAC and ATLAS

- Search scripts find best blocking factors
- Code generation system customizes the code
- Result is portable high performance
- Complex software system
- Hard to maintain and/or modify (the numerical code is controlled indirectly)

C++ to the Rescue!

- With *template meta-programming* techniques, variable degrees of unrolling can be directly expressed
- Made possible by integer template parameters

```
template <class T, int M>
class X {
    ...
};
```

The Fixed Algorithm Size Template (FAST) Library

- Essentially STL for fixed (at compile time) size computations
- A combination of generic programming with template meta-programs
- Suitable for small sized, performance critical kernels
- Demonstrates that extra abstraction levels do not hinder performance

Comparison of STL and FAST

```
// STL
int len = 4;
int* x = new int(len); int* y = new int(len);
fill(x, x+len, 1); fill(y, y+len, 3);
std::transform(x, x+len, y, y, plus<int>());

// FAST
const int LEN = 4;
int* x = new int(LEN); int* y = new int(LEN);
fill(x, x+LEN, 1); fill(y, y+LEN, 3);
fast::transform(x, cnt<LEN>(), y, y, plus<int>());
```

Definition of fast::transform()

- Recursion is used instead of loops
- Recursion depth is fixed and each call becomes inlined

```
template <int N, class InIter1, class InIter2,
          class OutIter, class BinOp>
OutIter
transform(InIter1 in1, cnt<N>, InIter2 in2,
          OutIter out, BinOp binary_op) {
    *out = binary_op (*in1, *in2);
    return transform(++in1, cnt<N-1>(), ++in2,
                    ++out, binary_op);
}
```

Basic Linear Algebra Instruction Set (BLAIS)

- Linear algebra kernels for fixed sized computations.
- Complete expansion results in no loops. Just as good as hand coded unrolling.
- Presents a simple and elegant interface.
- Simple implementation layered on the Fixed Algorithm Size Template (FAST) library.
- Template metaprograms can be elegant!

Definition of BLAIS mult()

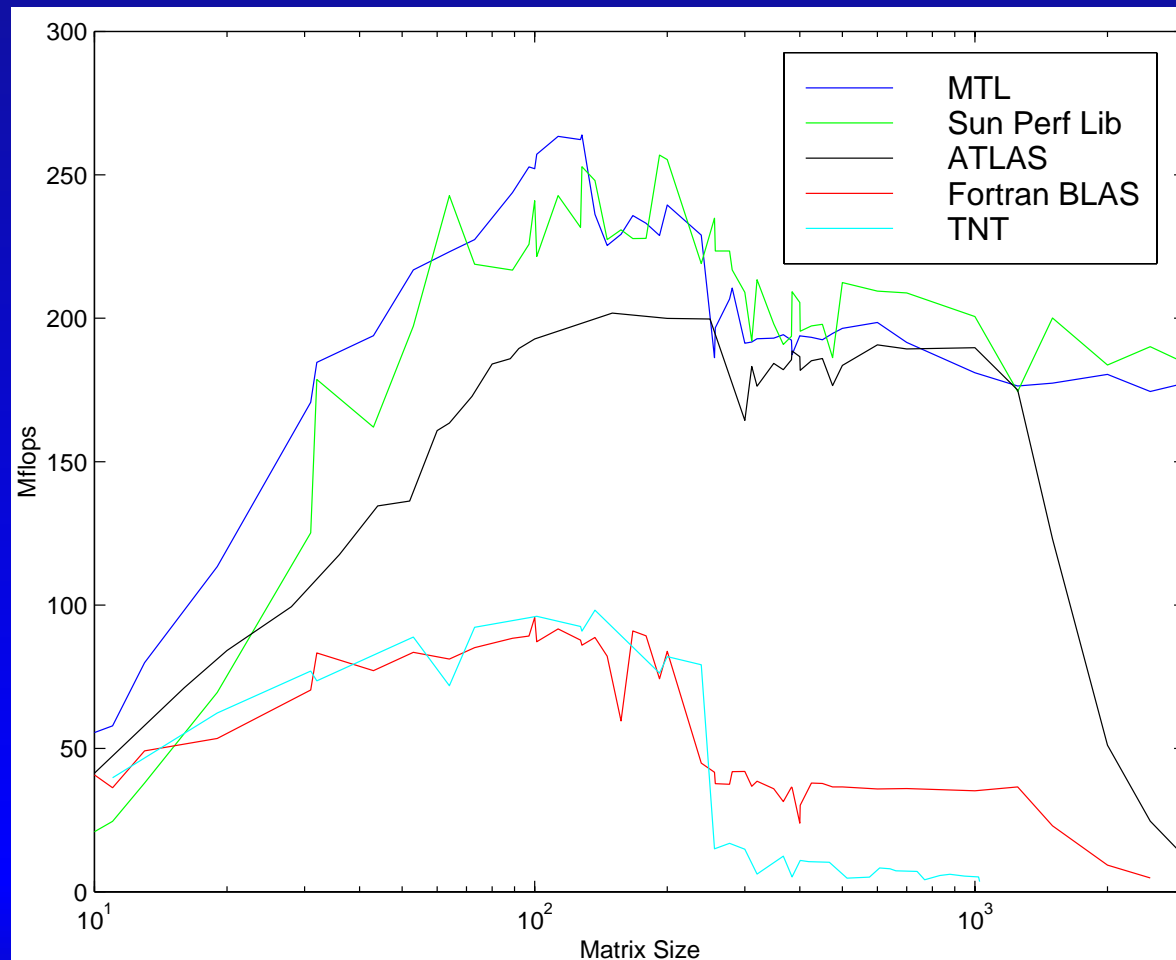
```
// General Case
template <int M, int N>
struct mult {
    template <class ColIter, class IterX,
              class IterY>
    mult(ColIter col_iter, IterX x, IterY y) {
        add<M>(scl((*col_iter).begin(), *x),
              y);
        mult<M, N-1>(++col_iter, ++x, y);
    }
};

// N = 0 Case
...
```

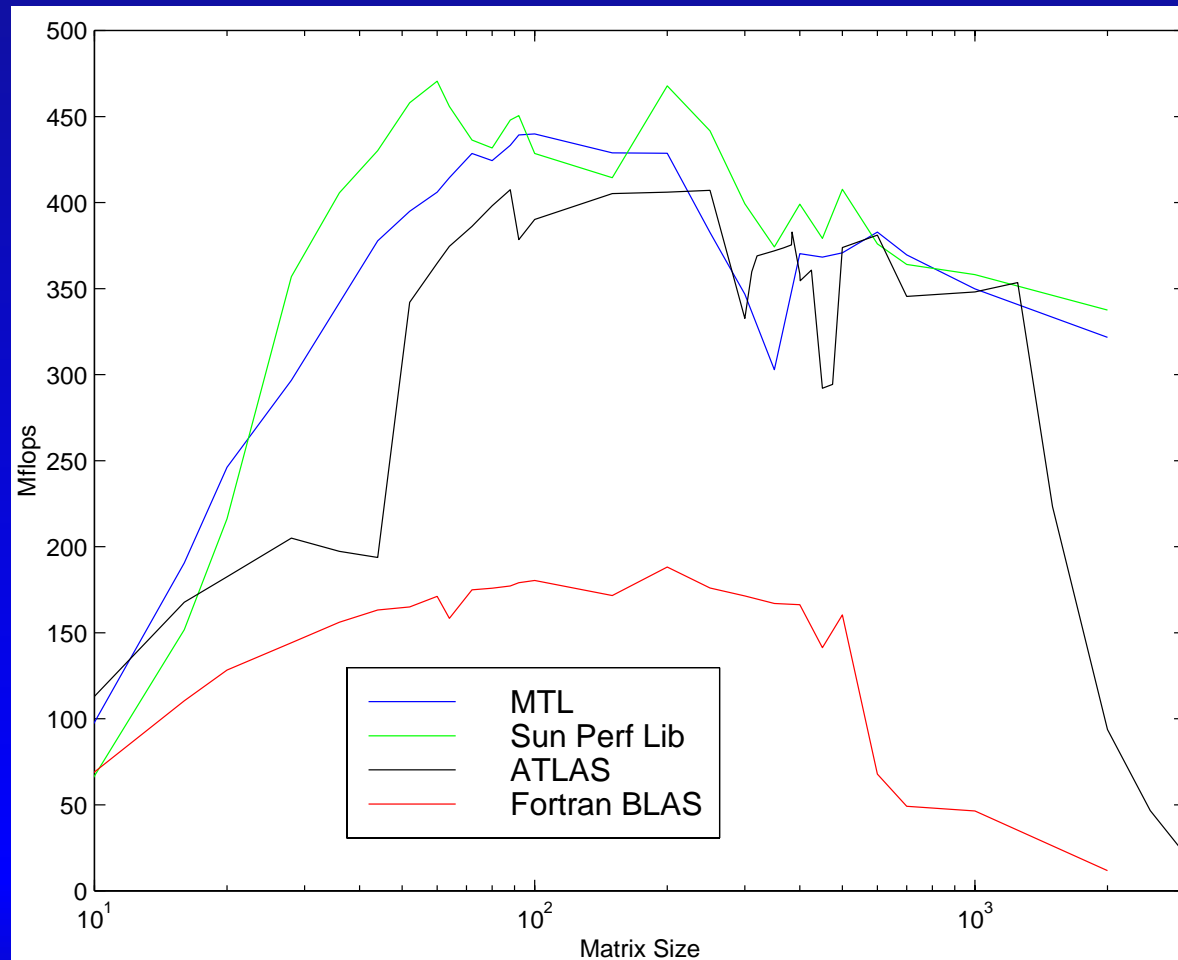
Recursive Matrix Product Using BLAIS

```
void mult(MatA& A, MatB& B, MatC& C) {
    while (A_k != A.end_rows()) {
        while (B_j != B.end_columns()) {
            MatC::Block Cblock = *C_kj;
            while (B_ji != (*B_j).end()) {
                blais::mult(*A_ki, *B_ji, Cblock);
                ++B_ji; ++A_ki;
            } // cleanup K left out
            ++B_j; ++C_kj;
        } // cleanup N left out
        ++A_k; ++C_k;
    } // cleanup M left out
}
```

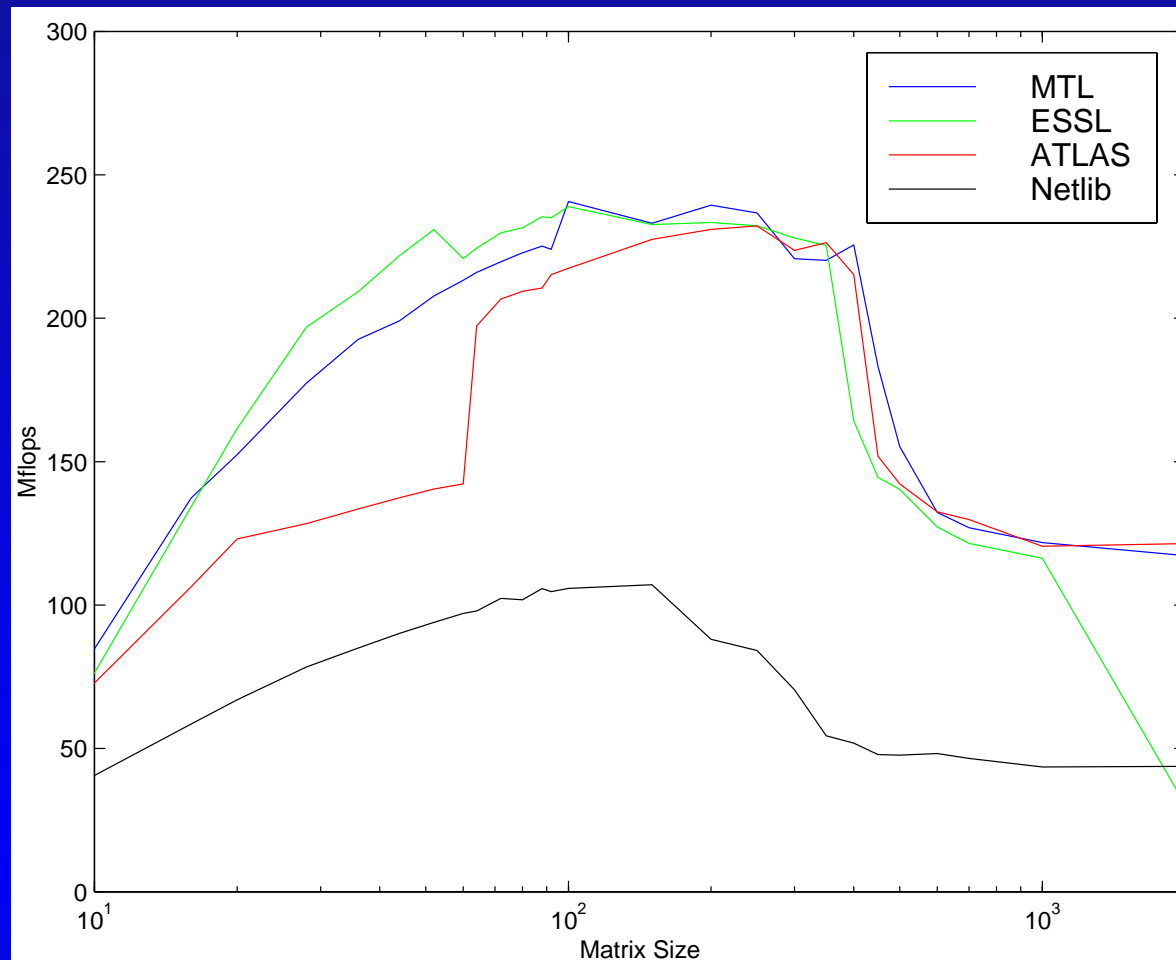
Dense Matrix-Matrix Performance (UltraSPARC 170E)



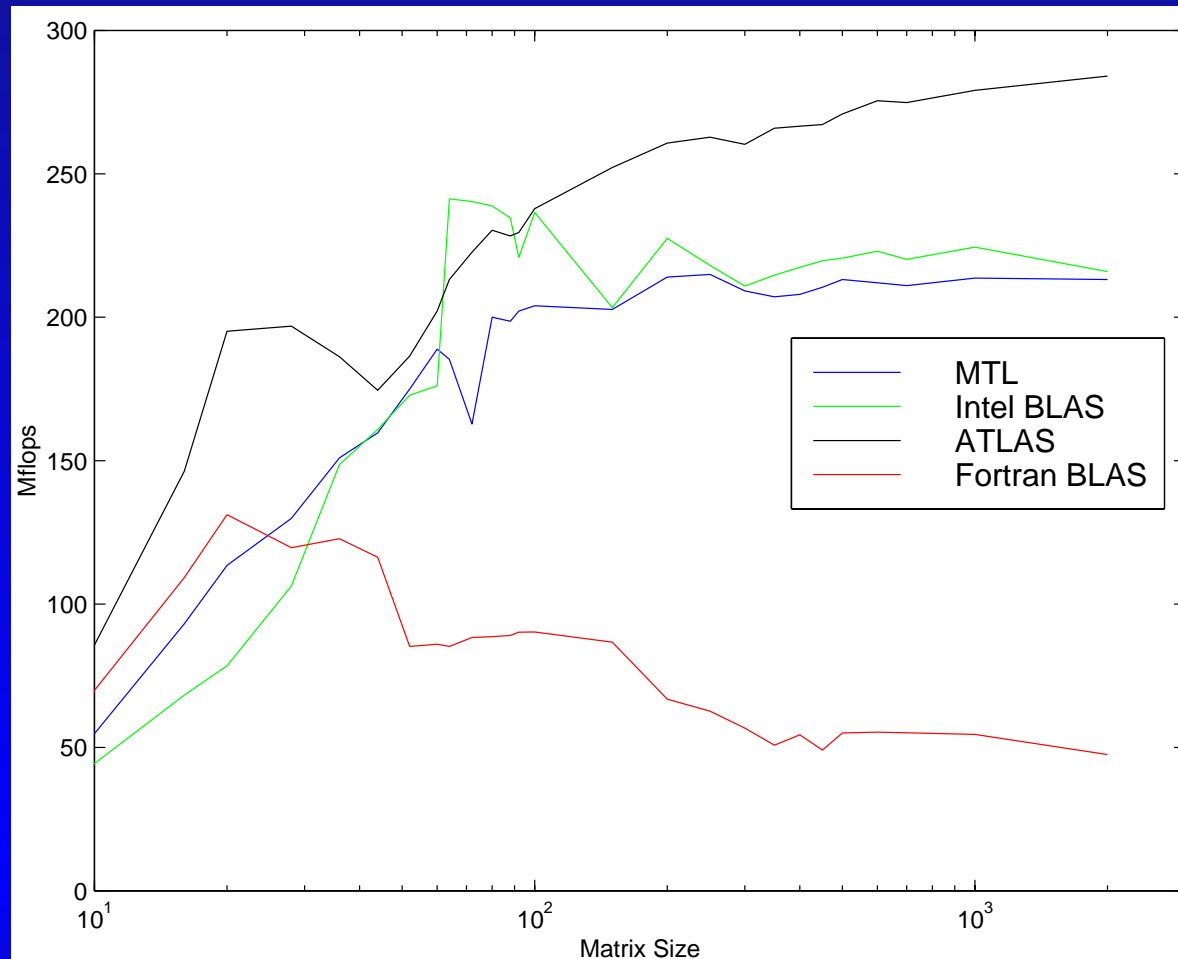
Dense Matrix-Matrix Performance (UltraSPARC 30)



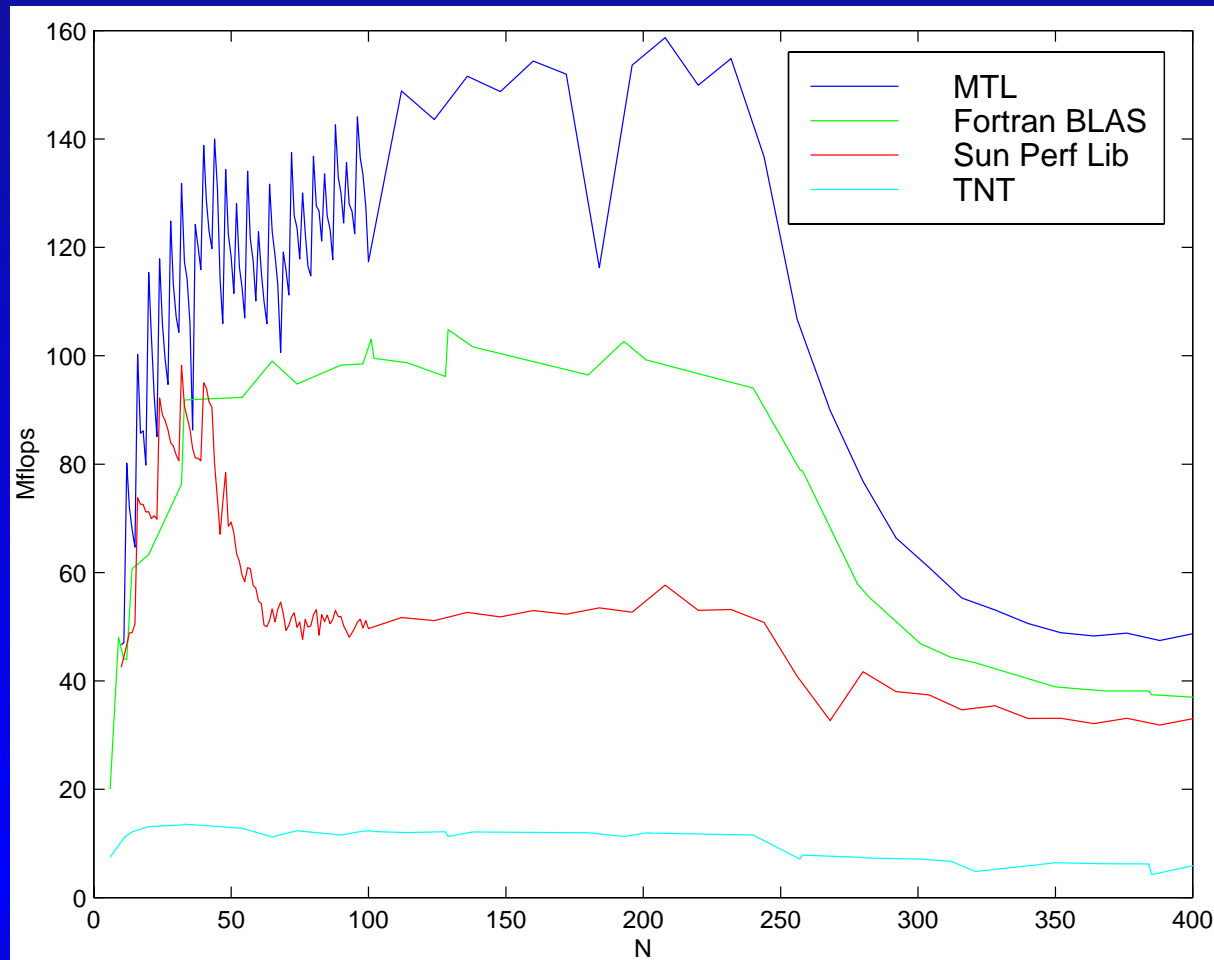
Dense Matrix-Matrix Performance (RS6000 590)



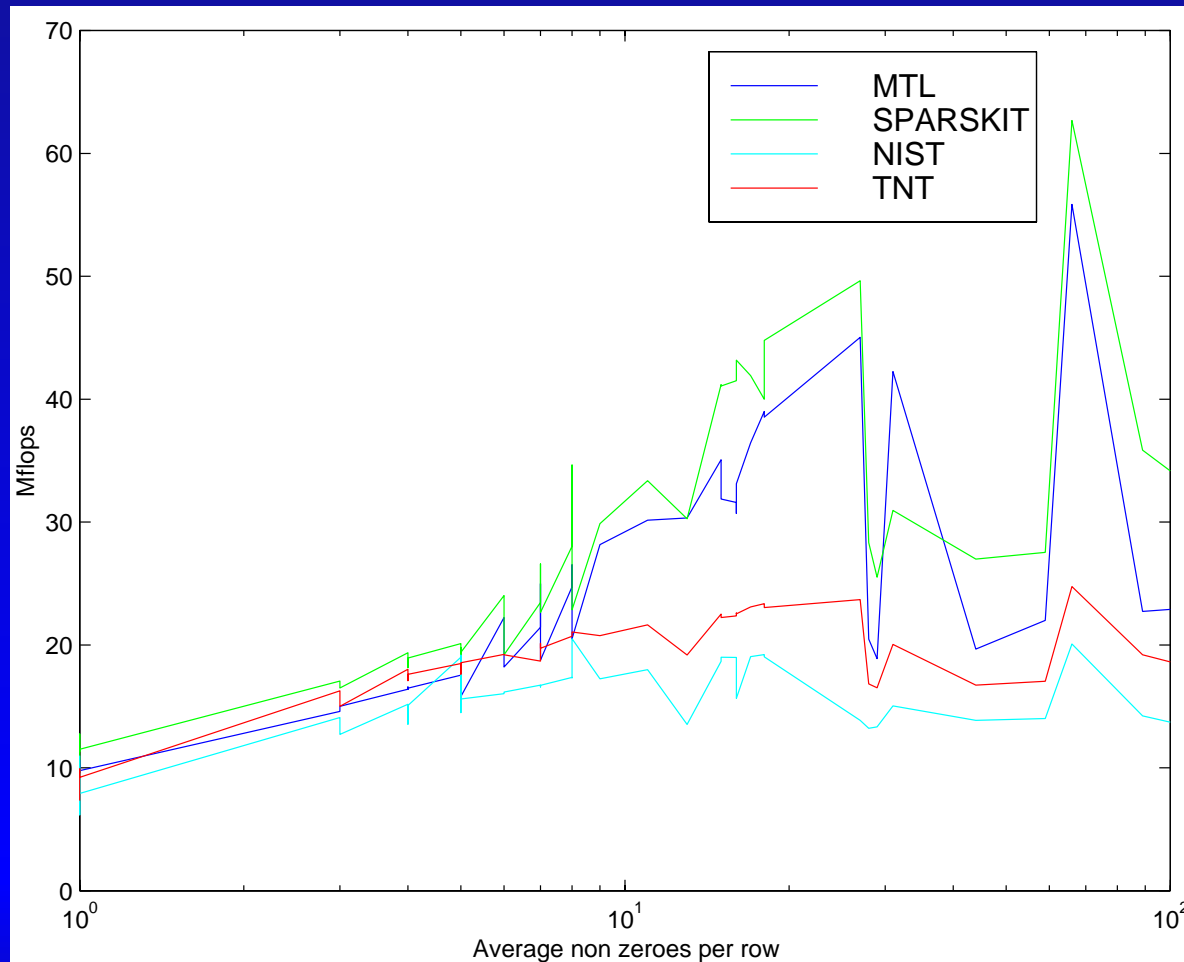
Dense Matrix-Matrix Performance (Pentium II 400Mhz)



Dense Matrix-Vector Performance (UltraSPARC 170E)



Sparse Matrix-Vector Performance (UltraSPARC 170E)



Conclusions:

What does genericity buy?

- Huge (multiplicative) functionality delivered with small (additive) effort
- Elegance
- Code re-use
- Performance

Conclusions: Performance

- Performance is a solved problem and is not a valid argument for the use of primitive programming languages
- Performance portability is a solved problem. Libraries do not need to be supplied by vendors.
- Performance portable, functionally comprehensive libraries can be developed with reasonable scale of effort