Generic Programming for High Performance Numerical Linear Algebra

Jeremy Siek and Andrew Lumsdaine

Department of Computer Science and Engineering

University of Notre Dame
Overview

1. Introduction & Motivation
2. Generic Programming
3. Generic Algorithms for Linear Algebra
4. The MTL Algorithms
5. The MTL Components
6. High Performance
7. Conclusion
Introduction & Motivation

- Scientific software can benefit from software engineering methodologies
  - Development
  - Maintenance
- Perpetual interest in using C++ (e.g.) in scientific computing
- Common perception: Abstraction is the enemy of performance
Introduction & Motivation

- C++ can be used effectively in scientific computing (with concomitant software engineering benefits)

- Generic programming has some particular benefits in this domain

- Follow the theme of STL

- Keep high-performance always in mind
Combinatorial Explosion

- Four precision types
- Several dense storage types
- A multitude of sparse storage types
- Row and column oriented matrices
- Scaling and striding
Combinatorial Explosion

- Unnecessary artifact of certain programming languages
- Algorithm expression also includes data type information
- Not necessary in certain other languages (most notably, C++)
Generic Programming

- Algorithms can be expressed independently of data storage formats
- Define standard *interfaces* for data storage components
- *Iterators* are the interface between *containers* and algorithms
- E.g., The Standard Template Library
- High performance linear algebra *is* amenable to generic programming
Iterator Bridge Between Algorithms and Containers

Algorithm \( \overset{\text{Iterator}}{\longleftrightarrow} \) Container
Example of Generic Programming

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

// how it is used:
vector<double> x(10, 1.0);
double sum = accumulate(x.begin(), x.end(), 0.0);
Generic Algorithms for Linear Algebra

- Extend the generic style of programming to domain of linear algebra

- A matrix 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.
The MTL Generic Algorithms

- Encompasses BLAS functionality
- A *single* algorithm typically used for all matrix and numeric types
- Index-less algorithms
- Sparse and dense algorithms unified
- Transpose, scaling, and striding handled by adapters, not the algorithm
Index-less Algorithms

- Iterate from \texttt{begin()} to \texttt{end()} of a vector.
- Iterate from \texttt{begin_rows()} to \texttt{end_rows()} (or 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.

```cpp
for(j = i->begin(); not_at(j, i->end()); ++j)
    tmp += *j * x[j.index()];
```
A Generic Matrix-Vector Multiply

template <class Matrix, class IterX, class IterY>
void matvec::mult(Matrix A, IterX x, IterY y) {
    typename Matrix::row_2Diterator i;
    typename Matrix::RowVector::iterator j;
    for (i = A.begin_rows();
        not_at(i, A.end_rows()); ++i) {
        typename Matrix::PR tmp = y[i.index()];
        for (j = i->begin(); not_at(j, i->end()); ++j)
            tmp += *j * x[j.index()];
        y[i.index()] = tmp;
    }
}
Transpose, Scaling, and Striding

- Matrix and vector adapters
- An adapter wraps up an object and modifies its behavior

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

matvec::mult(trans(A),
              scale(x, alpha),
              stride(y, incy));
```
The MTL Components

- Iterators
- 1-D Containers
- 2-D Containers
- Orientation Adapter: Row and Column
- Shape Adapter: Banded, Triangle, Symmetric

```
triangle<row<array2D<dense1D<double>>>>, lower>
```
The MTL Iterators

- For sparse vectors
- For dense vectors
- Strided iterator adapter
- Scaled iterator adapter
- Block iterator
The MTL 1-D Containers

- dense1D similar to STL's vector class
- sparse1D index-value pairs
- compressed1D separate index and value arrays
- scaled1D adapter class
- strided adapter class

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` adapter class

```
triangle<row<array2D<dense1D<double>>>,lower>
```
The MTL Orientation Classes

- **Row Orientation**
  - Maps row to *major*
  - Maps column to *minor*

- **Column Orientation**
  - Maps column to *major*
  - Maps row to *minor*

- All 2-D methods and typedefs are mapped.

```
triangle<row<array2D<dense1D<double>>>>, lower>
```
The MTL Shape Adapters

- banded, triangle
- symmetric, hermitian

triangle<row<array2D<dense1D<double>>>>, lower
High Performance with C++

- Explicit unrolling and blocking in C++ using the BLAIS.

- Use a good optimizing compiler to remove layers of abstraction: lightweight object optimization and inlining.

- Follow a set of coding guidelines to ensure the above optimizations can be made, and double check the intermediate C code.

- Don't interfere with backend compiler unrolling and scheduling.
Dense Matrix-Matrix Performance (UltraSPARC 170E)
## Dense Matrix-Matrix Performance

**(RS6000 590)**

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Mflops MTL</th>
<th>Mflops ESSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing performance comparison between MTL and ESSL](image)
Dense Matrix-Vector Performance

(UltraSPARC 170E)
Sparse Matrix-Vector Performance
(UltraSPARC 170E)
Conclusion

- High performance linear algebra
- Comprehensive (sparse, dense, etc.) and orthogonal
- Only 25,000 lines of code
- 150,000 lines for Fortran BLAS