Overview

1. Introduction
2. Previous Work
3. The FAST Library
4. The BLAIS Library
5. Matrix-Matrix Multiply Code with BLAIS
6. Performance Results
7. Conclusion
Introduction

- Basic Linear Algebra
- High Performance
- Portability
High performance in lower levels the most critical

BLAIS n' FAST for the inner-loops.

<table>
<thead>
<tr>
<th>Scientific Applications</th>
<th>LAPACK</th>
<th>Gnatlab</th>
<th>Iterative Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Template Library</td>
<td></td>
<td></td>
<td>BLAIS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>FAST</td>
</tr>
</tbody>
</table>
How to Achieve High Performance

- Maximize use of high-performance architecture features
  - Hierarchical memory
  - Instruction pipelining

- General goals
  - Provide better data locality (spatial and temporal)
  - Deliver code to the backend compiler that is easy to optimize (at the instruction scheduling and register allocation level)
How to Achieve High Performance

- Data locality
  - Loop ordering
  - Loop blocking
- Instruction stream
  - Loop unrolling
  - Register-level blocking
  - Instruction order
Why is Portable High Performance Difficult?

- Blocking sizes are machine dependent
- Number of blocking levels is machine dependent
- Loops cannot be used for register blocking
  (unrolling/blocking must be done “by hand” for best pipeline performance and register usage)
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 metaprogram* techniques, **variable** degrees of unrolling can be directly expressed.

- Made possible by integer template parameters

```cpp
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 metaprograms
- 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. The recursion depth is fixed and each call becomes inlined.

```cpp
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

- 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 layed on the Fixed Algorithm Size Template (FAST) library.
- Template metaprograms can be elegant!
The BLAIS Implementation

- Maps generic FAST algorithms into mathematical operations.
- Functionality similar to Level 1, 2, and 3 BLAS.
- Each level implemented in terms of the previous level.
Example usage of BLAIS vecvec::add()

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

// add<4>() expands at compile time to:
y[0] += a * x[0];
y[1] += a * x[1];
y[2] += a * x[2];
y[3] += a * x[3];
Definition of BLAIS vecvec::add()

- Implementation is simply a call to `fast::transform()`.

- Used the constructor to get nice syntax.

```cpp
template <int N>
struct add {
    template <class Iter1, class Iter2>
    add(Iter1 x, Iter2 y) {
        typedef ... T;
        fast::transform(x, cnt<N>(), y, y, plus<T>());
    }
};
```
The BLAIS Matrix-Vector Multiply

- We need to extend the generic style of programming to matrices.
- Think of a matrix as a *container of containers*.
- Use *iterators* and *2-dimensional iterators* to traverse the matrix. In the following slide the *2-dimensional iterator* will be a *column iterator*. 
Definition of BLAIS matvec::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) {
        vecvec::add<M>(scl((*col_iter).begin(),*x), y);
        mult<M, N-1>(++col_iter, ++x, y);
    }
};

// N = 0 Case
...
Building a Matrix-Matrix Multiply

High Performance Issues for $C = A \ast B$:

- Hierarchical blocking to make the best use of the memory.
- Copy small pieces of matrix $C$ to registers to reduce memory I/O and avoid aliasing problems.
- Copy larger portions of $A$ to reduce level-1 cache conflict misses.
Building a Matrix-Matrix Multiply

Portability Issues:

- No need for a code generation system, use BLAIS.
- Collect changes in one place.
- Changes should be easy for user to make.
Recursive Matrix-Matrix Multiply Setup

template <class MatA, class MatB, class MatC>
void matmat::mult(MatA& A, MatB& B, MatC& C) {
    MatA::RegisterBlock<4,1> A_L0;
    MatB::RegisterBlock<1,2> B_L0;
    MatC::RegCopyBlock<4,2> C_L0;
    MatA::CopyBlock A_L1(16,128);
    MatB::Block B_L1(128,16);
    MatC::Block C_L1(16,16);
    matmat::__mult(block(block(A, A_LO), A_L1),
                   block(block(B, B_L0), B_L1),
                   block(block(C, C_L0), C_L1));
}
Recursive Matrix-Matrix Multiply

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

Matrix Multiply Kernel Performance

![Graph showing matrix multiply kernel performance comparison between BLAIS and Sun Perf Lib.](image-url)
Matrix-Matrix Multiply Performance

(UltraSPARC 170E)
Matrix-Matrix Multiply Performance

(RS6000 590)

![Chart showing matrix-matrix multiply performance for MTL and ESSL libraries. The x-axis represents matrix size, and the y-axis represents Mflops. The chart illustrates the performance comparison between MTL and ESSL over different matrix sizes.]
Conclusion

- Portable high performance can be made easy!
- The right software abstractions aid performance.
- BLAIS fits well into the MTL generic framework.
- C++ is expressive enough, no need for code generation systems.