Outline

1. HPC == numerical linear algebra?
   - The role of linear algebra in HPC
   - Numerical linear algebra

2. Dense Matrix Linear Algebra
   - BLAS
   - LAPACK
   - ATLAS

3. Matrix-matrix products
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
Outline

1. **HPC == numerical linear algebra?**
   - The role of linear algebra in HPC
   - Numerical linear algebra

2. **Dense Matrix Linear Algebra**
   - BLAS
   - LAPACK
   - ATLAS

3. **Matrix-matrix products**
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
Perhaps the largest single group of HPC applications are devoted to solving problems described by differential equations.

- In these problems there is typically a domain, which may be space, time, both of these, or something else altogether.
- The domain is *discretized*, resulting in a system of equations that must be solved.
- These equations are often linear so techniques from numerical linear algebra are used to solve them.
- When the equations are nonlinear we may be able to *linearize* them (treat them as linear on a portion of the domain) to get the solution part-by-part.
HPC != numerical linear algebra?

Some of the problems that can lead to differential equations and linear systems as well as others that don't involve differential equations can be approached in completely different ways.

Two important examples:

1. simulation by Monte Carlo methods
2. dealing with data: searching and sorting
As we’ve seen in the case of matrix-matrix multiplication, standard “textbook” algorithms are often not the most efficient implementations.

In addition, they may not be the most “correct” in the sense of keeping errors as small as possible.

Since it’s so important to “get it right” and “get it right fast,” it’s nearly always advisable to use numerical libraries of proven code to handle most linear algebra tasks.

Today we’ll focus on a few important libraries. There are many others, and you should always look for available appropriate libraries that can help as part of tackling any programming problem.
Outline

1. HPC == numerical linear algebra?
   - The role of linear algebra in HPC
   - Numerical linear algebra

2. Dense Matrix Linear Algebra
   - BLAS
   - LAPACK
   - ATLAS

3. Matrix-matrix products
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
Key operations on vectors include:

- copy one vector to another
- swap two vectors
- scale a vector by a constant
- add a multiple of one vector to another
- inner product
- 1-norm of a vector
- 2-norm of a vector
- find maximal entry in a vector
Matrix-vector operations

Key operations between matrices and vectors include:

- general matrix-vector multiplication
- sparse matrix-vector multiplication
- rank-1 update (adds a scalar multiple of $xy^T$ to a matrix)
Key operations between pairs of matrices include:

- general matrix-matrix multiplication
- symmetric matrix-matrix multiplication
- sparse matrix-matrix multiplication
Outline

1. HPC == numerical linear algebra?
   - The role of linear algebra in HPC
   - Numerical linear algebra

2. Dense Matrix Linear Algebra
   - BLAS
   - LAPACK
   - ATLAS

3. Matrix-matrix products
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
The BLAS are routines that provide standard building blocks for performing basic vector and matrix operations:

- the Level 1 BLAS perform scalar, vector and vector-vector operations,
- the Level 2 BLAS perform matrix-vector operations, and
- the Level 3 BLAS perform matrix-matrix operations.

Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software.

The BLAS homepage is [http://www.netlib.org/blas/](http://www.netlib.org/blas/).
BLAS description

- The name of each BLAS routine (usually) begins with a character that indicates the type of data it operates on:
  - S  Real single precision.
  - D  Real double precision.
  - C  Complex single precision.
  - Z  Complex double precision.
- Level 1 BLAS handle operations that are $O(n)$ data and $O(n)$ work
- Level 2 BLAS handle operations that are $O(n^2)$ data and $O(n^2)$ work
- Level 3 BLAS handle operations that are $O(n^2)$ data and $O(n^3)$ work
Level 1 BLAS

Level 1 BLAS are designed for operations with $O(n)$ data and $O(n)$ work.

Some BLAS 1 subprograms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xCOPY</td>
<td>copy one vector to another</td>
</tr>
<tr>
<td>xSWAP</td>
<td>swap two vectors</td>
</tr>
<tr>
<td>xSCAL</td>
<td>scale a vector by a constant</td>
</tr>
<tr>
<td>xAXPY</td>
<td>add a multiple of one vector to another</td>
</tr>
<tr>
<td>xDOT</td>
<td>inner product</td>
</tr>
<tr>
<td>xASUM</td>
<td>1-norm of a vector</td>
</tr>
<tr>
<td>xNRM2</td>
<td>2-norm of a vector</td>
</tr>
<tr>
<td>IxAMAX</td>
<td>find maximal entry in a vector</td>
</tr>
</tbody>
</table>

Notice the exception to the naming convention in the last line. BLAS functions returning an integer have names starting with I so the datatype indicator is displaced to the second position.

BLAS 1 quick reference: [http://www.netlib.org/blas/blasqr.ps](http://www.netlib.org/blas/blasqr.ps)
Level 2 BLAS are designed for operations with $O(n^2)$ data and $O(n^2)$ work.

Some BLAS 2 subprograms

- xGEMV: general matrix-vector multiplication
- xGER: general rank-1 update
- xSYR2: symmetric rank-2 update
- xTRSV: solve a triangular system of equations

A detailed description of BLAS 2 can be found at http://www.netlib.org/blas/blas2-paper.ps.
Level 3 BLAS are designed for operations with $O(n^2)$ data and $O(n^3)$ work.

Some BLAS 3 subprograms

- `xGEMM` general matrix-matrix multiplication
- `xSYMM` symmetric matrix-matrix multiplication
- `xSYRK` symmetric rank-$k$ update
- `xSYR2K` symmetric rank-$2k$ update

A detailed description of BLAS 3 can be found at [http://www.netlib.org/blas/blas3-paper.ps](http://www.netlib.org/blas/blas3-paper.ps).
BLAS are written in FORTRAN

- Calling BLAS routines from FORTRAN is relatively straightforward.
- FORTRAN stores two-dimension arrays in column-major order. Many BLAS routines accept a parameter named LDA which stands for the *leading dimension of A*.
- LDA is needed to handle strides between elements common to a single row of A.
- In contrast, C and C++ store two-dimensional arrays in row-major order. Rather than the leading dimension, the trailing dimension specifies the stride between elements common to a single column.
- Calling vector-only BLAS routines from C is relatively straightforward.
- One must work with transposes of matrices in C/C++ programs to use the BLAS matrix-vector and matrix-matrix routines.
A version of the BLAS has been written with routines that can be called directly from C/C++.

The CBLAS routines have the same name as their FORTRAN counterparts except all subprogram and function names are lowercase and are prepended with `cblas`.

The CBLAS matrix routines also take an additional parameter to indicate if the matrix is stored in row-major or column-major order.
Even with the matrix transpose issue, it’s fairly easy to call the Fortran BLAS from C or C++. 

1. Find out the calling sequence for the routine. Man pages for BLAS routines can be looked up on-line. You may also find the BLAS quick reference (http://www.netlib.org/blas/blasqr.pdf or http://www.netlib.org/lapack/lug/node145.html) helpful.

2. Create a prototype in your C/C++ code for the function. The function name should match the Fortran function name in lower-case and have an appended underscore.

3. Optionally create an interface function to make handling the differences between how C and Fortran treat arguments easier.
Example: BLAS calling routine DGEMM() from C

DGEMM() performs the operation

\[ C \leftarrow \alpha \text{op}(A)\text{op}(B) + \beta C \]

where op(A) can either be A or \( A^T \). The Fortran calling sequence is

\[ \text{DGEMM}(\text{TRANSA}, \text{TRANSB}, M, N, K, \alpha, LDA, B, LDB, \beta, C, LDC) \]

**TRANSA,TRANSB**: (CHARACTER*1) ’N’ for no transpose, ’T’ or ’Y’ to transpose

- **M**: (INTEGER) number of rows in C and A
- **N**: (INTEGER) number of columns in C and B
- **K**: (INTEGER) number of columns in A and rows in B

**ALPHA, BETA**: (DOUBLE PRECISION) scale factors for \( AB \) and C

**LDA, LDB, LDC**: (INTEGER) leading dimensions of A, B, and C
Example: BLAS calling routine DGEMM() from C

The corresponding prototype in a C or C++ file would look like

```c
#if defined(__cplusplus)
extern "C" {
#endif

void dgemm_(char * transa, char * transb, 
    int * m, int * n, int * k, 
    double * alpha, double * a, int * lda, 
    double * b, int * ldb, 
    double * beta, double * c, int * ldc);

#if defined(__cplusplus)
}
#endif
```

- Each parameter is a pointer since Fortran does not support *pass-by-value*.
- The `extern "C"` is necessary to prevent C++ name mangling.
- Using `#if defined(__cplusplus)` ensures this code will work with both C and C++. 

Finally, it’s often helpful to write an interface function that handles the different way Fortran and C parameters are treated:

```c
void dgemm(char transa, char transb, int m, int n, int k,
    double alpha, double* a, int lda,
    double* b, int ldb,
    double beta, double* c, int ldc)
{
    dgemm_(&transa, &transb, &m, &n, &k, &alpha, a, &lda,
            b, &ldb, &beta, c, &ldc);
}
```

- Notice that this allows us to pass `transa` and `transb` and the other scalar values as literals, whereas without this interface routine we would not be able to.
- Check out `02-mem-heir/matrix_prod.c` in the class repository [http://www.math-cs.gordon.edu/hg/CPS343/](http://www.math-cs.gordon.edu/hg/CPS343/) for a complete example.
Outline

1. HPC == numerical linear algebra?
   - The role of linear algebra in HPC
   - Numerical linear algebra

2. Dense Matrix Linear Algebra
   - BLAS
   - LAPACK
   - ATLAS

3. Matrix-matrix products
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
The **Linear Algebra PACKage** is a successor to both LINPACK and EISPACK.

- LINPACK is a library of FORTRAN routines for numerical linear algebra and written in the 1970s.
- EISPACK is a library of FORTRAN routines for numerical computation of eigenvalues and eigenvectors of matrices and was also written in the 1970s.
- One of the LINPACK authors, Cleve Moler, went on to write an interactive, user-friendly front end to these libraries called **MATLAB**.
- LINPACK and EISPACK primarily make use of the level 1 BLAS
- LAPACK largely replaces LINPACK and EISPACK but takes advantage of level 2 and level 3 BLAS for more efficient operation on computers with hierarchical memory and shared memory multiprocessors.

The LAPACK homepage is [http://www.netlib.org/lapack/](http://www.netlib.org/lapack/).
LAPACK is written in FORTRAN 90, so calling from C/C++ has the same challenges as discussed for the BLAS.

A version called CLAPACK exists that can be more easily called from C/C++ but still has the column-major matrix access issue.
Outline

1. HPC == numerical linear algebra?
   - The role of linear algebra in HPC
   - Numerical linear algebra

2. Dense Matrix Linear Algebra
   - BLAS
   - LAPACK
   - ATLAS

3. Matrix-matrix products
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
ATLAS

ATLAS stands for **Automatically Tuned Linear Algebra Software** and consists of the BLAS and some routines from LAPACK.

- The key to getting high performance from the BLAS (and hence from LAPACK) is using BLAS routines that are tuned to each particular machine’s architecture and compiler.
- Vendors of HPC equipment typically supply a BLAS library that has been optimized for their machines.
- The ATLAS project was created to provide similar support for HPC equipment built from commodity hardware (e.g. Beowulf clusters).

ATLAS Performance Example

The graph below shows GFLOPS vs matrix dimension for both the GSL (GNU Scientific Library) BLAS and the ATLAS BLAS supplied with Ubuntu 10.04 and running on a Lenovo T410 laptop. We see close to a 6-fold increase in performance for the larger matrix sizes.
Outline

1. HPC == numerical linear algebra?
   - The role of linear algebra in HPC
   - Numerical linear algebra

2. Dense Matrix Linear Algebra
   - BLAS
   - LAPACK
   - ATLAS

3. Matrix-matrix products
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
The rowwise matrix-matrix product for $C = AB$ is computed with the pseudo-code below.

```
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $c_{ij} = 0$
        for $k = 1$ to $n$ do
            $c_{ij} = c_{ij} + a_{ik}b_{kj}$
        endfor
    endfor
endfor
```

Notice that both $A$ and $C$ are accessed by rows in the innermost loop while $B$ is accessed by columns.
The graph below shows GFLOPS vs matrix dimension for a rowwise matrix-matrix product. The drop-off in performance between $N = 300$ and $N = 400$ suggests that all of $B$ can fit in the processor’s cache when $N = 300$ but not when $N = 400$.

- A $300 \times 300$ matrix using double precision floating point values consumes approximately 700 KB.
- A $400 \times 400$ matrix needs at least 1250 KB.
- The cache on the machine used for this test is 2048 KB.

Note: This graph was generated on the previous generation Minor Prophets machines.
1. **HPC == numerical linear algebra?**
   - The role of linear algebra in HPC
   - Numerical linear algebra

2. **Dense Matrix Linear Algebra**
   - BLAS
   - LAPACK
   - ATLAS

3. **Matrix-matrix products**
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
Key idea: minimize cache misses

If we partition $A$ and $B$ into appropriate sized blocks

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}, \quad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

then the product $C = AB$ can be computed as

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}$$

where each of the matrix-matrix products involves matrices of roughly $1/4$ the size of the full matrices.
We can exploit this idea quite easily with a recursive algorithm.

```
matrix function C = mult(A, B)
    if (size(B) > threshold) then
        C00 = mult(A00, B00) + mult(A01, B10)
        C01 = mult(A00, B01) + mult(A01, B11)
        C10 = mult(A10, B00) + mult(A11, B10)
        C11 = mult(A10, B01) + mult(A11, B11)
    else
        C = AB
    endif
return C
```
Recursive block-oriented matrix-matrix product

The graph below shows GFLOPS vs matrix dimension for both a rowwise product and a recursive block-oriented matrix-matrix product.

- Performance is improved across the board, including for smaller matrix sizes.
- Performance is relatively flat - no degradation as matrix sizes increase.
Finally, we compare the results just shown with similar results using the default BLAS routines supplied with Ubuntu 12.04 and ATLAS BLAS created on the Minor Prophets.

- **ATLAS BLAS** performance is far superior to either the standard BLAS or the recursive block routine.