CS294 / CS194 - Homework Assignment 3
FFT and Dense Linear Algebra
Due date: October 28, 2011

October 16, 2011

1 Specific Instructions

You are to code up and check in

FFT1DRecursive.cpp Your implementation of the recursive version of the discrete FFT

FFTW1D.cpp Your class that calls the FFTW library to perform the discrete FFT

dgemm-naive.cpp Your version of square matrix matrix multiply (the triple nested loop).

dgemm-blas.cpp Your implementation of dgemm that calls the cblas library.

naive.out naive_opt.out blas.out The screen output from the three dgemm runs you are told to perform in section 3.

report.pdf The responses to specific question in Section 2 and Section 3.

2 FFT

1. Implement the recursive algorithm for the power-of-2 FFT as a derived class from the interface class `FFT1D`. The algorithm is given as follows. If we define

$$\mathcal{F}_N(f)_k \equiv \sum_{j=0}^{N-1} f_j z_N^{jk} \ , \ z_N = e^{\pm 2\pi\iota/N}, k = 0, \dots N-1$$

then we compute, for $k = 0, \dots, \frac{N}{2} - 1$

$$\begin{aligned}\mathcal{F}_N(f)_k &= \mathcal{F}_{N/2}(\mathcal{E}(f))_k + z_N^k \mathcal{F}_{N/2}(\mathcal{O}(f))_k \\ \mathcal{F}_N(f)_{k+\frac{N}{2}} &= \mathcal{F}_{N/2}(\mathcal{E}(f))_k - z_N^k \mathcal{F}_{N/2}(\mathcal{O}(f))_k\end{aligned}$$

where $\pm = -$ for the forward transform, $\pm = +$ for the inverse transform, and

$$\mathcal{E}(f)_j = f_{2j} \ , \ \mathcal{O}(f)_j = f_{2j+1} \ , \ j = 0, \ldots, \frac{N}{2} - 1$$

Implement this by recursively calling the derived class on the even and odd points, until you reach the case $N = 2$. The two-point FFT is given by

$$\mathcal{F}_2(f)_0 = f_0 + f_1 \ , \ \mathcal{F}_2(f)_1 = f_0 - f_1$$

2. Test your classes derived from `FFT1D` in the `FFT1DTest.cpp` driver. The makefile target is `test1d`, and the executable name is `test1d.ex`. You should be able to make it as it stands when you check it out of svn, except it will only run the "BRI" case. When you have both of your classes working, compile them using the optimized compiler flags, and run them using the Unix shell `time` command, i.e.
   `>time ./test1d.ex`
   for the cases $log_2(N) = 20, 21, 22$. The UNIX `time` command reports three numbers. `real, user, sys` Report the `user` times as part of your writeup of your assignment. `user` will exclude the program time spent waiting for inputs and outputs.

3. Once your codes have passed the 1D tests, test the FFT1D implementations (including BRI, which is already implemented) in the `FFTMDTest.cpp` test code, which is currently set up for 3D problems. Integrate them into the test code following the example using BRI that is already there. The makefile target is `testMD`, and the executable is called `testMD.ex` When the code is running correctly for small problems, recompile with optimization on, and run for $M = log_2(N) = 7$ (and 8 if you have enough memory on your laptop) using `time`, for all three FFT1D implementations.

Note that there is a function in `PowerItoI.{H,cpp}` that raises an integer to a positive integer power. You may find it useful.

3 Matrix Multiply

You will implement dense general matrix matrix multiply for two column-wise stored dense matrices A and B. This data looks like the data layout we described for the float* indexing in Homework 1, except we will now be using double precision (double not float). The result is stored into another Matrix C. These will be square matrices, and a few dozen matrix sizes will be executed and then checked for correctness. A and B are initialized with random data. The main routine computes an estimate for the MFlops/s (ie, millions of floating point operations per second, a measure of raw compute performance)

1. Implement your own triple nested loop version of dense matrix multiply and put it in the file dgemm-naive.cpp. it will contain one function declared as

   `void square_dgemm( int n, double *A, double *B, double *C ).`

   Compile the 'naive' makefile target and execute and capture the output in a file named 'naive.out' which you check into the repo

2. Change the compiler flags from the default '-g -Wall' flags to '-O3'. ie, turn on compiler optimization. build 'naive' again, and produce a '`naive_opt.out`' output.

3. Implement a version of this function in a file named `dgemm-blas.cpp` with a call to `cblas` third party library. Build the '`blas`' makefile target. run the code and create a '`blas.out`' output for your problem to submit.

Bonus: estimate what percent of peak performance these three implementations are achieving of your processor's peak floating-point performance. Include a description of your processor in your estimate and your calculation.