CS294 - Homework Assignment 3
FFT and Dense Linear Algebra
Due date: October 31,2015

October 10, 2024

1 FFT

1. Implement the recursive algorithm for the Cooley-Tukey power-of-2 FFT as a derived class `FFT1DRecursive` 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

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

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. Implement a wrapper class `FFT1DW` for the open source FFT solver package FFTW. You will need to download and configure and install FFTW from source by downloading the source code from http://fftw.org/ and following the instructions. I recommend issuing the `make check` command after the `make` command to verify you have working executable code.

   The `configure` script takes an optional `--prefix` option which lets you control where the installation will target. You will need to know this location when you modify the `Make.include` file I've given you.

3. Test your classes `FFT1DRecursive` (recursive implementation of Cooley-Tukey algorithm given above) and `FFTW1DW` (a wrapper around the `fftw` code), both 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 the `test1d` target as it stands when you check it out of git, except it will only run the "BRI" case. In order to run your other two cases in the 1D test driver, you will have to implement the two virtual functions. When you have both of your classes working, compile them using the optimized compiler flags `-O3` for the cases $log_2(N) = 20$. The test program will output the time and the error for computing the transform, followed by the inverse transform, of a Gaussian; and the extent to which a single Fourier mode is captured exactly by the forward transform. Not the differences in the running time, and in the error in the Gaussian, for the various cases / implementations.

4. Once your codes have passed the 1D tests, test the `FFT1DBRI` and `FFTW1D` implementations 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 `test3D.ex`. You should be able to build `test3D.exe` as it comes out of git, which will call the BRI case. When the code is running correctly for small problems, recompile with optimization on, and run for $M = log_2(N) =$ 7,8 (and 9 if you have enough memory on your laptop) for both FFT1D implementations.

5. Now you will write your own benchmark driver program. An example of a benchmark driver is given in the Matrix Multiply section of this homework. You will create a new target in the `testFFT/GNUMakefile` for your benchmark program and name the target and the executable `fftBenchmark.exe`. You will hard code it to run in 3D and use the `FFTMD` class used in the `testMD` exercise. It does not have to test correctness, as we will already have verified correctness. The benchmark should run through `M=[4, 5, 6, 7, 8, 9]` and run with FFTW and BRI 1D solvers. You will include the output of this benchmark output.

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

2 Matrix Multiply

You will implement dense general matrix matrix multiply for two column-wise stored dense matrices A and B.

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j} \ , \ 1 \leq i,j \leq n$$

This data looks like the data layout we described for the multidimensional array indexing in Homework 1. 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 triply-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 the `cblas` third party library. Build the '`blas`' makefile target. For Macs, the makefile in this directory contain the appropriate compiler directives to access the tuned blas. Run the code and create a '`blas.out`' output for your problem to submit.