Project 1: Using Eigen

Due March 9th, 2018

Instructions

Include source files and a `CMakeLists.txt` file that compiles your code into a single executable.

Project Overview

In this project we will be learning how to use a third-party library in our own programs. Specifically, we will be using the Eigen library, which allows us to easily perform linear algebra operations. Additionally, we will be experimenting with standard library functions available to us from the `random` system header.

The Eigen 3 library can be found at `http://eigen.tuxfamily.org/index.php?title=Main_Page` or at the GitHub mirror `https://github.com/eigenteam/eigen-git-mirror`.

Your project should have the following directory layout:

```
<project name>
  /src
    - source_file1.cpp (etc.)
    - header_file1.h (etc.)
    - main.cpp (Required!)
  /lib
    /eigen
      - // eigen source
  CMakeLists.txt
```

Project Instructions

Part 1: Getting and building Eigen (10 points)

Part one of the project consists of the following steps:

1. Download the Eigen source
2. Move the Eigen source into your project `lib` directory
3. Create a `CMakeLists.txt` file that finds the Eigen library and compiles `main.cpp`

Part 2: Using Eigen for linear algebra (60 points)

In this section we will build some small classes that utilize Eigen. For random number generation, use the `std::mt19937_64` generator.

Building a random matrix class (10 points)

```
// Constructor
RandomMatrix(size_t row, size_t col, int low, int high)

// Returns an Eigen matrix of size row x col, whose elements
// are randomly chosen from the interval [low, high)
Eigen::MatrixXf RandomMatrix::generate();
```

Building a random vector class (10 points)

```
// Constructor
RandomVector(size_t s, int low, int high)

// Returns an Eigen vector of size s, whose elements
// are randomly chosen from the interval [low, high)
Eigen::VectorXf RandomVector::generate();
```

Creating test functions (20 points)

We will create two test functions in this problem:

```
std::pair<double, double> time_matrix_multiplication(size_t n);
std::pair<double, double> time_vector_dot(size_t n);
```

The functions are largely the same. The functions will consturct a random $n \times n$ matrix (or vector) using the previously built classes as well as the same matrix (vector) using `std::vector<std::vector<int> >` (or `std::vector<int>` for the vector). The function should then perform multiplication (matrix-matrix, or vector dot product), compare that results are the same, and time how long it takes to perform the multiplication operation. The goal is to compare the performance of Eigen with a simple implementation based on `std::vector`. Each function should return a `std::pair<double, double>` containing the times of the Eigen mutliplication (first) and the `std::vector` multiplication (second).

As a refresher, we define the vector dot product as follows:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=0}^{n-1} x_i \cdot y_i$$

Matrix-matrix multiplication (for two $n \times n$ sized matrices) is defined as follows. The element of the i-th row and j-th column of the product matrix $C = AB$ is given by the formula

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} B_{k,j}$$

You should use the `std::chrono::steady_clock` to time your operations.

Comparing Eigen to std::vector (20 points)

In this part we will use the previously built comparison functions to carry out a small experiment comparing our matrix-matrix/vector dot product implementation against Eigen's. Run comparisons for matrix-matrix multiplication for $n \in \{10, 50, 100, 250, 500, 1000\}$. Vector dot product test should be run for $n \in \{100, 1000, 100000, 1000000\}$. Make sure to print the results in a well formatted and descriptive output. For example:

```
Matrix-matrix multiplication tests
size: 10
  Eigen: 6e-6 s
  std::vector: 6e-6 s

size: 50
  Eigen: 6e-5 s
  std::vector: 6e-4 s

size: 100
  Eigen: 6e-4 s
  std::vector: 6e-3 s
```

Save these results in file titled “unoptimized.txt”.

Now we are going to look at how we can improve the performance of our code by increasing the compiler optimization level. Set the compiler optimization to O3 and rerun the tests. Save the results in a file title “optimized.txt”. **Note**: you can redirect output from stdout using the `>` operator. For example, suppose you have a binary called `a.out`. You can capture all the output of `a.out` while it is running into text file named “test.txt” via the following snippet:

```
$ a.out > test.txt
```

Also note that test.txt does not need to exist prior to running the command; it will be created automatically.

Hints

The following headers will be useful:

- `<chrono>`
- `<random>`

`http://en.cppreference.com/w/` is an excellent resource for information on using the facilities provided by these headers.

Submission Instructions

Your submission should include the following:

1. `.txt` containing your output from the last part of the project.
2. A `CMakeLists.txt` file that compiles your code into a single executable

References

[1] Bjarne Stroustroup. *A Tour of C++*. Pearson Education, 2014