UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 4/16/2017
Problem Set #4 **Due:** 4/25/2017

1 Preliminaries

1.1 C++ Classes

You are expected to have a surface level understanding of C++ classes for this assignment. If you are still unclear on C++ classes after lecture, there is a Panopto tutorial available on the "Announcements" tab of the course webpage.

1.2 Pre-Processing

Although not part of the C++ language per se, the pre-processor allows you to programmatically manipulate the text of the program itself. That is, you can make changes to the program text – in an automated fashion – before it is actually compiled. One example we touched on was in the use of `assert()` for defensive programming.

When `assert()` is enabled, it executes a check of the expression that is passed to it – that is, it executes a small amount of code. In performance critical applications, this can steal cycles as well as prevent certain compiler optimizations. For production (release) versions of a program, we need to be able to remove the assertions.

One way to remove the assertions, of course, would be to go to all of your files and either delete or comment out all of the calls to `assert()`. This is a anti-solution: during development we need to be able to switch between debug and release versions of the code. We need to be able to switch *all* of the calls to `assert()` on and off in one fell swoop each time. To support this, calls to `assert()` can be "turned off" by the pre-processor if the macro `NDEBUG` is defined.

The pre-processor works in the following basic way. It takes your program as input and provides program text as an output. How this output is produced is controlled by the pre-processors own macro language – which is also embedded in the program text. Some of the statements you have already seen in your programs, such as `#include` are pre-processor statements. In fact, all statements beginning with `#` are pre-processor language commands.

Beyond just filtering input and output, the pre-processor has some sophisticated features for text manipulation. However, we are just going to look at some basic capabilities for including or excluding specific parts of your program before being passed to the compiler.

The command you have already seen from the pre-processor is `#include`, which is used to pull the text of one file into another file. For example, if your file `Vector.cpp` includes the file `Vector.hpp`

```
#include "Vector.hpp"

int main() {

  return 0;
}
```

the text that is passed to the compiler is the concatenation of the two files. That is, what is compiled when you invoke the compiler on `Vector.cpp` is the complete text of `Vector.hpp` with `Vector.cpp`, with the text of `Vector.hpp` inserted at the location of the `#include ''Vector.hpp''`. If you would like to see all of the text that is passed to the compiler after pre-processing, you can add the option "`-E`" to your compilation command. Be careful with this though – if you have included anything from the standard library you will get an enormous amount of text back. The statements `#include <iostream>` are not special – there is a file named `iostream` that is part of the standard library and its text is pulled in with that pre-processor directive.

Now, as we mentioned above, the pre-processor can be programmed by the user. To do this, we need to be able to do expected programming tasks, such as defining and testing variables and branching based on the tests. Since the pre-processing text and the program text are combined – and since the pre-processor manipulates the program text – it is important to distinguish between pre-processor commands and variables, and the program text and variables. The convention is to use `ALL_CAPS` for any pre-processor macros or variables.

Variables in the pre-processor are defined with the following syntax:

```
#define MY_VARIABLE sometext
```

This creates the macro (pre-processor variable) with the name `MY_VARIABLE` in the pre-processor namespace. Two things happen when a pre-processor macro is defined. First, whenever the pre-processor encounters the text `MY_VARIABLE` in the program text, it substitutes the defined text for that variable. In other words, the following transformation occurs.

```
#include <iostream>
using namespace std;

#define MY_GREETING "Hello World"
#define OP *

int main() {

  cout << MY_GREETING << endl;
  cout << "7 x 6 = " << 7 OP 6 << endl;

  return 0;
}
```

$\Longrightarrow$

```

using namespace std;



int main() {

  cout << "Hello World" << endl;
  cout << "7 x 6 = " << 7 * 6 << endl;

  return 0;
}
```

Note that the macros `MY_GREETING` and `OP` are replaced by *exactly* the text they are define to be. In this case, that even includes the quotation marks in `MY_GREETING`.

Branching in the pre-processor is controlled by a family of `#if` directives: `#if`, `#ifdef`, and `#ifndef`, which test the value of a compile-time expression, whether a macro is defined, or whether a macro is not defined, respectively. In response to evaluating an `#if` the pre-processor doesn't execute one branch of pre-processor code or another, rather it sends one stream of your program text or another to the compiler.

Defensive Programming One standard use of the branching capabiliites of pre-processor is to make sure that the text from any give header files is only placed once into the text stream to the compiler. This can easily happen when multiple headers include each other and/or include multiple headers from the standard library. In that case, even though the headers might be `#include`d, we only insert their text once. The following is standard technique. For any header file, the following pre-processor directives are used (assume the header file is `Matrix.hpp`):

```
#ifndef MATRIX_HPP   // if the macro MATRIX_HPP is not defined, include the following text
#define MATRIX_HPP  // First, define the macro


#endif  // The program text up to the matching #endif is what is included
```

You should make a habit of always protecting your header files in this way.

As you might expect, there is an `#else` to go along with `#if`.

```
#include <iostream>

int main() {
```

```
#ifdef BAD_DAY
  std::cout << "Today is a bad day'' << std::endl;
#else
  std::cout << "Today is a good day" << std::endl;
#endif

  return 0;
}
```

In this example, the compiler will get the first branch of text if the macro `BAD_DAY` is defined, otherwise it will get the second branch. **NB:** With `#ifdef`, it does not matter what the value of the macro is. The test is only whether the macro exists or not. It is perfectly acceptable to `#define` a macro with no value.

In fact, when we want to disable `assert()`, we just need to `#define` the macro `NDEBUG`, we don't need to give it any particular value. But, since the pre-processor just processes your program text and sends it to the compiler, the `NDEBUG` macro *must* be defined before the `#include <cassert>` statement.

But this raises almost the same scalability issue we mentioned before. If we want to globally remove assert, we need to have the `NDEBUG` macro defined when processing our program files. One way to do this would be to edit each of the files and insert (or remove) `#define NDEBUG` in every one. This is impractical for all but the smallest of programs (and maybe even not then).

There is an essential feature of the C++ compiler that solves this problem, namely the `-D` option. This option passes a macro (with or without a defined value) to the prep-processor. In particular, you can pass `NDEBUG` to your programs this way (without ever having to change the program):

```
c++ -DNDEBUG main.cpp -o main.exe
```

You can give the macro a value by using `=`

```
c++ -DNDEBUG=1 main.cpp -o main.exe
```

but for `NDEBUG` the definition, not any value, is what turns on or turns off `assert()`.

Automation In keeping with the course philosophy of "automate anything repetitive", the place to take advantage of incorporating (or not) `NDEBUG` definitions is in your `Makefile`. However, a problem with a familiar feature appears. Namely, if we want to globally turn on or turn off `assert()`, we have to make a sweep through the `Makefile` and add it or remove it from every production rule. If we truly want to be able to enable or disable `assert()` with the flip of a switch (as it were), we need to just be able to change it in *one place* but have the effect be global.

To programmatically control the automation that is introduced by `make`, the `make` program also has its own macro language. Fully using those capabilities can quickly turn into "Deep Magic" and we want to avoid that, but there are some basic features that enable the "edit once - change everywhere" behavior we are looking for.

In particular, `make` allows you to define macros that are expanded within the body of the `Makefile`. Two very common macros that you might use are to define which compiler you want to use, and to define which flags you want to pass to the compiler. For example, to make the programs `dot5893` and `vectorNorm` we might have the following intermediate rules in the `Makefile`:

```
dot583.o: dot583.cpp amath583.hpp
        c++ -Wall -g -std=c++11 -c dot583.cpp -o dot583.o

amath583.o: amath583.cpp amath583.hpp
        c++ -Wall -g -std=c++11 -c amath583.cpp -o amath583.o

vectorNorm.o: vectorNorm.cpp amath583.hpp
        c++ -Wall -g -std=c++11 -c vectorNorm.cpp -o vectorNorm.o
```

And one can easily imagine having many more production rules for any number of files. Now, suppose we wanted to introduce `NDEBUG`, or change the compiler we are using, or switch between release levels of C++? Manually, we would have to edit every line, doing it completely consistently, and not introducing any errors. Or, we can use a macro and do it once. Using a `Makefile` macro is straightforward. You define it with `=` and use it with `\$`. If we used the standard approach in the above `Makefile`, it would look like this:

```
CXX      = c++
CXXFLAGS = -Wall -g -std=c++11

dot583.o: dot583.cpp amath583.hpp
        $(CXX) $(CXXFLAGS) -c dot583.cpp -o dot583.o

amath583.o: amath583.cpp amath583.hpp
        $(CXX) $(CXXFLAGS) -c amath583.cpp -o amath583.o

vectorNorm.o: vectorNorm.cpp amath583.hpp
        $(CXX) $(CXXFLAGS) -c vectorNorm.cpp -o vectorNorm.o
```

Now, if we wanted to add `NDEBUG` to disable `assert()`, we simply change the definition of `CXXFLAGS`:

```
CXXFLAGS = -Wall -g -std=c++11 -DNDEBUG
```

We can do the same with switching from debug mode to release mode

```
CXXFLAGS = -Wall -O3 -std=c++11 -DNDEBUG
```

(Note the use of `-O3` rather than `-g`.) There are a number of conventions used in naming and use of `Makefile` macros – and a number are pre-defined for you. Those wanting to further leverage the automation capabilities of `make` are encouraged to consult the on-line references given on the course web site.

Just for Ninjas For the truly lazy (in a good way), you will quickly notice that there is *still* alot of repetition in the `Makefile`. All of the production rules have the same pattern

```
        $(CXX) $(CXXFLAGS) -c <something>.cpp -o <something>.o
```

Compile "something.cpp" to create "something.o". In large programs with large `Makefile`s, keeping these all consistent could benefit from automation. To automate pattern-based production rules, `make` has some "magic" macros that pattern match for you.

```
CXX      = c++
CXXFLAGS = -Wall -g -std=c++11

dot583.o: amath583.hpp
dot583.o: dot583.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@

amath583.o: amath583.hpp
amath583.o: amath583.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@

vectorNorm.o: amath583.hpp
vectorNorm.o: vectorNorm.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

A few things to note here before we get to our final `Makefile`. First, dependencies don't have to all be on one line. In the above, we express the dependencies for the object files in multiple lines, one dependency per line. The lines that then have production rules are executed when any of the dependencies are not met.

```
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

In this production rule, the macro `$<` means the file that is the dependency – the .cpp file – and `$@` means the target.

There is one last thing to clean up here. We *still* have a repetitive pattern – all we did was substitute the magic macros in. But every production rule still has:

```
something.o : something.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

There is one more pattern matching mechanism that `make` can use – implicit rules. We express an implicit rule like this:

```
%.o : %.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

This basically the rule with "something" above – but we only need to write this rule *once* and it covers all cases where something.o depends on something.cpp. We still have the other dependencies (on headers) to account for – but this can also be automated – google for "makedepend".

At any rate, the Makefile we started with now looks like this:

```
CXX      = c++
CXXFLAGS = -Wall -g -std=c++11

%.o : %.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@

dot583.o: amath583.hpp
amath583.o: amath583.hpp
vectorNorm.o: amath583.hpp
```

This will handle *all* cases where something.o depends on something.cpp.

NB: Unless otherwise instructed, you do *not* have to use any of the advanced features of `make` in your assignments as long as your programs correctly build for the test script. However, keep in mind that these mechanisms were developed to save time and decrease mistakes while programming.

cpuinfo As discussed in lecture (7B), the SIMD/vector instruction sets for Intel architectures have been evolving over time. Although modern compilers can generate machine code for any of these architectures, the chip that a program is running on must support the instructions generated by the compiler. If the CPU fetches an instruction that it cannot execute, it will throw an illegal instruction error.

The `cpuid` machine instruction can be used to query the microprocessor about its capabilities. How to issue these queries and how to interpret the results is explained (e.g.) in the wikipedia entry for cpuid. If you use the following command with docker

```
$ docker run amath583/cpuid
```

you will get a listing that shows a selection of available capabilities on your own machine. Run this command and check the output. What level of SIMD/vector support does your machine provide?

The particular macros to look for are anything with "SSE", "AVX", "AVX2", or "AVX512". These support 128-, 256-, 256-, and 512-bit operands, respectively. What is the maximum operand size that your computer will support?

Intel Power Gadget One of the factors impacting CPU performance is CPU clock speed. In today's computer systems, the clock speed is adjusted dynamically depending on processor load. Download and install the Intel Power Gadget from `http://intel.ly/2plTqJl`. Open it and while it is running, execute some of the matrix-matrix programs from previous assignments (pick some that run for at least a few seconds). What is the highest clock rate that your computer supports? What is the lowest (when nothing compute-intensive is running)?

Peak Performance Peak floating point performance for one core is the performance your computer would achieve if all it did was repeat the most compute-intensive operation once per clock cycle. For instance, if the clock rate of your computer is 2.5GHz and it supports SSE, peak performance would be 5.0 FLOPS – 2 FLOPS per clock cycle. If, in addition to SSE your CPU supports fused multiply-add, it would be able to perform 4FLOPS per clock cycle for a peak rate of 10.0 FLOPS. Based on the clock rate of your computer and its level of support for SIMD/Vector instructions, what is its peak rate for a single core?

1.3 Sparse Matrix Computation

We have been estimating floating point performance for dense matrix-matrix product by counting the number of floating-point operations and the total number of operations in the inner loop of a matrix-matrix product routine. We have been measuring the performance of matrix-matrix product by timing a run of it and dividing the total number of floating point operations (as determined by the three nested loops) by that time.

In lecture we saw that performing sparse matrix-vector product was much faster (in terms of elapsed time) than dense matrix-vector product. However, we did not characterize the rate at which the floating point operations were being performed.

For a sparse matrix, we are storing only the non-zero elements of the matrix. Thus, the looping constructs for performing sparse matrix-vector product depend on the number of non-zero elements, not on the matrix dimensions. The number of floating point operations are easy to count: For each non-zero element, we have to look up a column index, look up a row index (in the case of COO), look up y, look up x, perform a multiply, an add, and then store y. The result is two floating point operations in seven total operations per non-zero element.

2 Warm Up

2.1 The make Command

To verify, and gain some familiarity with, the operation of `Makefile` and pre-processor macros, work through the following examples.

Create the following program and name it `fail.cpp`:

```
#include <iostream>
#include <cassert>

int main() {

  assert(1 == 0);

  std::cout << "Hello World" << std::endl;

  return 0;
}
```

Notice that the assertion `(1 == 0)` will always be false and so the assertion will fail. On failure, the program will abort, and will abort before it prints the hello message. Try compiling and running that program with no particular compiler options and verify that it aborts before printing the message.

Next compile the program with the `-DNDEBUG` option.

```
$ c++ -DNDEBUG fail.cpp
```

What happens when you run the resulting `a.out`?

Finally, create a test `Makefile` that looks like the following:

```
CXX      = c++
CXXFLAGS = -Wall -g -std=c++11

fail:  fail.cpp
        $(CXX) $(CXXFLAGS) fail.cpp -o fail

clean:
        /bin/rm -f fail fail.o a.out
```

Preliminary question – when you type `make fail`, what gets created? What happens when you run that program?

Finally, modify the above `Makefile` so that it compiles a program that ignores the `assert()`. Do that without changing the production rule (the compilation line after the dependency rule for `fail`).

2.2 Timing

As discussed in lecture – if we are going to achieve "high performance computing" we need to be able to measure performance. Performance is the ratio of how much work is done in how much time – and so we need to measure (or calculate) both to quantitatively characterize performance.

To measure time, in lecture we also introduced a `Timer` class (available on the "Schedule" tab of the course website).

```
class Timer {
private:
  typedef std::chrono::time_point<std::chrono::system_clock> time_t;

public:
  Timer() : startTime(), stopTime() {}

  time_t start()   { return (startTime = std::chrono::system_clock::now()); }
  time_t stop()    { return (stopTime  = std::chrono::system_clock::now()); }
  double elapsed() { return
  std::chrono::duration_cast<std::chrono::milliseconds>(stopTime-startTime).count(); }

private:
  time_t startTime, stopTime;
};
```

To use this timer, you just need to `#include "Timer.hpp"`. To start timing invoke the `start()` member, to stop the timer invoke the `stop()` member. The elapsed time between the start and stop is reported with `elapsed()`.

To practice using the timer class, write and compile the following program.

```
#include <iostream>
using namespace std;

#include "Timer.hpp"

int main() {
  long loops = 1024L*1024L*1024L;

  Timer T;
  T.start();
  for (long i = 0; i < loops; ++i)
    ;
  T.stop();
  cout << loops << " loops took " << T.elapsed() << " milliseconds" << endl;

  return 0;
}
```

First, to get a baseline, compile it with no optimization at all. On my laptop, the 1G loops above took about 2 seconds. If your computer takes too long or too short, you can adjust the loop value (multiply it by 2 for example, or change one of the 1024 values into 512). What value does your computer give when timing this loop? How many milliseconds per loop? Note that the empty statement ";" in the loop body just means "do nothing."

Second, let's look at how much optimizing this program will help. Compile the same program as before, but this time use the `-O3` option. How much did your program speed up? Does this make sense? If you are unsure about the answer you are getting here, start a discussion on Piazza. Try to have this discussion sooner rather than later, as you will need some of the information gained for later in this assignment.

2.3 Abstraction Penalty and Efficiency

One question that arises as we continue to optimize, e.g., matrix multiply is: how much performance is available? The performance gains we saw in class were impressive, but are we doing well in an absolute sense? To flip the question around, and perhaps make it more specific: We are using a fairly deep set of abstractions to give ourselves notational convenience. That is rather than computing linear offsets from a pointer directly to access memory, we are invoking a member function of a class (recall `operator()()` is just a function. Then from that function we are invoking another function in the `vector<double>` class – `operator[]()`. And there may even be more levels of indirection underneath that function. Calling a function involves a number of operations, saving return addresses on the stack, saving parameters on the stack, jumping to a new program location – and then unwinding all of that when the function call has completed. When we were analyzing matrix-matrix product in lecture, we were assuming that the inner loop just involved a small number of memory accesses and floating point operations. We didn't consider the cost we might pay for having all of those function calls – calls we could be making at every iteration of the multiply function. If we were making those – or doing anything extraneous – we would also be measuring those when timing the multiply function. And, obviously, we would be giving up performance. The performance loss due to the use of programing abstractions is called the *abstraction penalty*.

One can measure the difference between achieved performance vs maximum possible performance as a ratio – as *efficiency*. Efficiency is simply

$$\frac{\text{Achieved performance}}{\text{Maximum performance}}$$

Let's write a short program to measure maximum performance – or at least measure performance without abstractions in the way.

```
  long N = 1024L;
  double a = 3.14, b = 3.14159, c = 0.0;
  Timer T;
  T.start();
  for (long i = 0; i < N*N*N; ++i) {
    c += a * b;
  }
  T.stop();
```

To save space, I am just including the timing portion of the program. In this loop we are multiplying two doubles and adding to doubles. And we are doing this N^3 times – exactly what we would do in a matrix-matrix multiply.

Time this loop with and without optimization. What happens? How can this be fixed? Again, this is a question I would like the class to discuss as a whole and arrive at a solution.

2.4 The Matrix Code

For this assignment you will be provided with a number of source code files, as well as a Makefile, in the Collect It Dropbox that you should put into a subdirectory `ps4`. The code for the `Matrix` class will be in `Matrix.cpp` and `Matrix.hpp`. Note that the files will contain most of the examples that we showed in lecture, but you need to use the Collect It version of `Matrix.hpp` instead of the webpage version. There will also be a driver program `bench.cpp` that when built (see the Makefile) will create an executable that will run performance tests for a variety of matrix sizes and algorithms (as specified on the command line). You should be able to deduce how it functions based on inspection of the code as well as simply building it and running it. Try some different sizes and different algorithms. In general you should keep the `-O3` flag turned on or you will be waiting a long time for the programs to complete.

Note that in the `Matrix.cpp` file we have provided a function for filling a matrix with random numbers as well as a function for zeroing out a matrix.

NB: I recommend that you take a quick tour through the code and familiarize yourself with some of the contents.

2.5 Additional Optimization Flags

When compiled with "-Ofast -march=native" on a machine supporting AVX2, the assembler for the inner loop of `hoistedTiledMultiply2x2()` looks like the following:

```
        vmovupd         (%rbx), %ymm5
        vmovupd         32(%rbx), %ymm6
        vmovupd         (%rsi), %ymm7
        vmovupd         8(%rsi), %ymm8
        vmovupd         32(%rsi), %ymm9
        vmovupd         40(%rsi), %ymm10
        vfmadd231pd         %ymm5, %ymm7, %ymm3
        vfmadd231pd         %ymm5, %ymm8, %ymm4
        vmovupd         -32(%rdi), %ymm5
        vmovupd         (%rdi), %ymm11
        vfmadd231pd         %ymm7, %ymm5, %ymm2
        vfmadd231pd         %ymm8, %ymm5, %ymm1
        vfmadd231pd         %ymm6, %ymm9, %ymm3
        vfmadd231pd         %ymm6, %ymm10, %ymm4
        vfmadd231pd         %ymm9, %ymm11, %ymm2
        vfmadd231pd         %ymm10, %ymm11, %ymm1
        addq        $64, %rdi
        addq        $64, %rsi
        addq        $64, %rbx
        addq        $-8, %rcx
```

The operation `vfmadd231pd` is a fused multiply-add operation. Since the operands start with "ymm", they are 256-bit operands (four doubles). Thus, each `vfmadd231pd` represents 8 floating point operations. Assuming a 3.5GHz clock rate, the peak floating point rate for this loop would be 8 floating point operations per cycle at 3.5×10^9 cycles per second — or 28GFLOPS.

Of course, and as we can see in the above example, we aren't only doing floating point operations. The loop includes a number of data movement operations ("vmov"). We can count the total number of clock cycles used for this section of the code by assuming each machine operation — each line of assembler — consumes one clock cycle. In this case we have 20 total instructions with 64 FLOPS. The practical peak floating point rate would be the number of FLOPS per loop divided by the total time per loop, i.e., 64 FLOPS $\div$ (20 operations $\div 3.5 \times 10^9$) or 11.2 GFLOPS.

Looking back at the source code for this routine, we see:

```
void hoistedTiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {
  for (int i = 0; i < A.numRows(); i += 2) {
    for (int j = 0; j < B.numCols(); j += 2) {
      double t00 = C(i,j);        double t01 = C(i,j+1);
      double t10 = C(i+1,j);      double t11 = C(i+1,j+1);

      for (int k = 0; k < A.numCols(); ++k) {
        t00 += A(i  , k) * B(k, j  );
        t01 += A(i  , k) * B(k, j+1);
        t10 += A(i+1, k) * B(k, j  );
        t11 += A(i+1, k) * B(k, j+1);
      }
      C(i,  j) = t00;  C(i,  j+1) = t01;
      C(i+1,j) = t10;  C(i+1,j+1) = t11;
    }
  }
}
```

In the inner loop we see four multiplies, four adds, four loads (unique) – or 12 total operations (assuming no loads or stores for `t00` et al). The four additional operations in the assembler output were due to bookkeeping needed for the loop itself so the actual count of operations in a loop may be higher than just the inner loop count. On the other hand, branch prediction and the ability to overlap certain types of operations my elide the overhead of the loop control code.

So in the loop we have 8 floating point operations and four loads. If no vectorization takes place, we could expect $3.5 \times 10^9 \times \frac{8}{12}$ or 2.3GFLOPS. With two-wide vectorization and four-wide, we could expect 4.6GFLOPS and 9.2 GFlops, respectively. And finally, with fused multiply add we could expect 18.4 GFlops.

Now, as we saw in the assembler above, we don't necessarily have just the four loads – and there is loop overhead. Instead we might have 8 FLOPS and 8 loads and 4 loop accounting operations. That would lower our expectation from 2.3GFLOPS to 1.4 GFLOPS for unvectorized code. With two-wide we would have 2.8GFLOPS, four four-wide, 5.6GFLOPS and four-wide fused multiply add, 11.2GFLOPS.

(**NB:** In general, you will have 2-wide with SSE, 4-wide with AVX, and 4-wide FMA with AVX2.)

3 Exercises

As a reminder, you are at risk for losing points for the following:

- Having `using namespace std;` in any header file that is included into other code
- Having warnings appear when your code is compiled with the `-Wall` flag
- Not including the corresponding libraries in your header files so that the header is complete

3.1 Abstraction Penalty

During lecture, I made the claim that implementation of the `Matrix` class with a linear array (as a `vector<double>` would be more efficient than an implementation as `vector<vector<double> >`. Let's verify if that is true or not.

As we discussed in lecture, all of the algorithms in the provided `Matrix.cpp` (see Warm Up section) use the *external* interface of the `Matrix` class. That means we can change the implementation in `Matrix.hpp` without changing any of the code in `Matrix.cpp`. **NB:** We don't have to change the code as written, but we do have to recompile it if the implementation changes.

To change the implementation of the `Matrix` class from a one-d to a two-d representation, we need to change a couple of things in the file `Matrix.hpp`. First, we have to change the implementation of `arrayData` to be a `vector<vector<double> >`. Once that is changed, we have to change all of the member functions that use it (since it has changed). That is, we have to write a new constructor and new `operator()()`. Note that the interfaces of these will not change, only how they access the internal element storage.

To compare the performance of one implementation to another, I want you to be able to switch from one representation to another, just my recompiling with a compiler flag. To do this, use the `#ifdef` technique above. You have a couple of options in how to approach this. You can either put the `#ifdef` around the entire `Matrix` class definition, or you can do it for the constructor and separately for the accessors and for the implementation. I would recommend the first way, but the choice is completely up to you.

The macro to select one implementation or the other must be called `ALTMATRIX`.

Deliverable A modified version of the `Matrix.hpp` file. When the matrix benchmarking code is compiled with the macro `ALTMATRIX` defined, it should compile the implementation based on `vector<vector<double> >` . Otherwise it should compile the version of the file given to you.

Example Switching between versions should look essentially like this:

```
#ifndef ALTMATRIX
  // our code
#else
  // your code
#endif // ALTMATRIX
```

The compilations would then be (without any of the other usual flags, which you would need in real life):

```
$ c++ -c Matrix.cpp -o Matrix.o

$ c++ -DALTMATRIX -c Matrix.cpp -o Matrix.o
```

This will respectively compile our version and your version.

Evaluation We will compile a test program similar to `bench.cpp` that uses your modified `Matrix.hpp` and the provided `Matrix.cpp`. We will verify performance and correctness, testing with a variety of matrix sizes, using matrix-matrix products. We will switch between implementations by defining (or not) the macro `ALTMATRIX`.

3.2 Matrix Vector

In the source code directory for this assignment you will also see there are the files `Vector.hpp` and `Vector.cpp`. You will also notice they have skeletons of a `Vector` class and supporting functions, along with comments about the portions of the code you need to fill in. For this part of the assignment, you are asked to complete their implementation.

Accessor Based on what we discussed about `const` and references and so forth relative to the accessors in the `Matrix` class, implement the necessary member functions for `operator()()` for the `Vector` class.

Multiply Given a matrix $A \in \mathbb{R}^{M \times N}$ and a vector $x \in \mathbb{R}^N$, we define the product of the matrix with the vector as

$$y_i = \sum_j A_{ij} x_j$$

The translation of that to a doubly-nested loop in code should be obvious (just one loop less than matrix-matrix). In the file `amath583.cpp`, complete the implementation of `matvec()` and `operator*()` (the analogy to mat-mat should make this straightforward). Your implementation of matrix-vector multiply should only use the external interface of `Vector`.

Finally, test your implementation for performance and correctness.

Deliverable Modified version of the `Vector.hpp` and `amath583.cpp` files. The `Vector.hpp` file should complete the implementation of the `Vector` class as described above. The `amath583.cpp` file should complete the matrix-vector multiplcation operation as described above.

Evaluation We will compile a test program similiar to `bench.cpp` that uses your modified `Vector.hpp` and `amath583.cpp` files, along with the provided `Matrix.cpp` and `Vector.cpp`. We will verify correctness of your implementations.

3.3 Tuning Matrix Vector (AMATH 583 Only)

As with matrix-matrix product, we would like to get better performance from matrix vector product than with just the naive algorithm. This exercise will investigate some potential mechanisms.

First, based on your matrix-vector function, create two matrix-vector functions `matvec_inner` and `matvec_outer` in `Vector.cpp`. The difference between these is that the first should iterate over the vector (over j in 3.2) in the *inner* loop, while the second should iterate over the vector (over j) in the *outer* loop. One of these should be identical to your existing `matvec` in `amath583.cpp`. Make sure to include their prototypes in `Vector.hpp`.

Modify `bench.cpp` to time your matrix-vector functions with the command line arguments `multMVinner` and `multMVouter`. This will likely require adding new `runBenchmark` and `benchmark` functions with the following prototypes:

```
void runBenchmark(function<void (const Matrix&, const Vector&, Vector&)> f,
        long maxsize);
double benchmark(int M, int N, long numruns,
        function<void (const Matrix&, const Vector&, Vector&)> f);
```

Next, create a function `matvec_student` in `Vector.cpp`, with the appropriate prototype in `Vector.hpp`. The only requirements on in it are that it produce a correct answer, that it not be identical to either `matvec_inner` or `matvec_outer`, and that it be faster than either `matvec_inner` or `matvec_outer`. You can provide conditions on this requirement in your written responses. Modify `bench.cpp` to time this new function with the command line argument `multMVstudent`.

Deliverables Modified versions of `Vector.cpp`, `Vector.hpp`, and `bench.cpp` as described above.

Evaluation We will compile both your modified `bench.cpp` and a test program similar to your modified `bench.cpp` that use your modified `Vector.hpp` and `Vector.cpp`. We will verify performance and correctness, testing with a variety of matrix sizes, using matrix-vector products.

3.4 Estimating Performance

Based on the example in 2.5, estimate the potential floating point performance for the "hoisted" benchmark for your CPU. Compare that with what you actually achieved. Repeat for "copyblockhoisted".

Deliverable A main program named "ps4bex1.cpp". When compiled, this program should print the following header:

```
Routine Clock CPUID Loop-ops Scalar 2-wide 4-wide 4-wide-fma Achieved
```

Then your program should print the name of the routine (hoisted in the first row, copyblockhoisted in the second). The following columns should contain the CPU speed of your processor, the maximum level of SIMD/vector support for your computer (SSE, AVX, AVX2, AVX512), the number of total operations in the loop (one line for 12 and one for 20). Following will be estimates for unvectorized, 2-wide, 4-wide, and 4-wide FMA code. Finally you should print the value that running bench with hoisted and copyblockhoisted achieved. Use a single space between items in a line and use endl to terminate each line.

NB: Use "-Ofast -march=native -DNDEBUG" as your optimization flags (rather than -O3).

Example With the scenario above your program would print

```
Routine Clock CPUID Loop-ops Scalar 2-wide 4-wide 4-wide-fma Achieved
hoisted 3.5E9 AVX2 12 2.3E9 4.6E9 9.2E9 18.4E9 7.1E9
hoisted 3.5E9 AVX2 20 1.4E9 2.8E9 5.6E9 11.2E9 7.1E9
copyblockhoisted 3.5E9 AVX2 12 2.3E9 4.6E9 9.2E9 18.4E9 13.1E9
copyblockhoisted 3.5E9 AVX2 20 1.4E9 2.8E9 5.6E9 11.2E9 13.1E9
```

3.5 Sparse Matrix Computation

Deliverable Complete the formula in the sparsebench.cpp file for computing the floating point performance (GFLOPS) for sparse matrix-vector product (see Section 1.3 above). Create a main program "ps4bex2.cpp" that prints the following formatted output:

```
Routine Clock CPUID Loop-ops Scalar 2-wide 4-wide 4-wide-fma Achieved
hoisted 3.5E9 AVX2 12 2.3E9 4.6E9 9.2E9 18.4E9 7.1E9
coo 3.5E9 AVX2 7 1.4E9 2.8E9 5.6E9 11.2E9 7.1E9
```

Here the coo line should show your estimates for peak achievable performance assuming 2 FLOPS and 7 total operations in the inner loop. The Achieved column should show the floating point ops per second using the formula you added to sparsebench.cpp.

3.6 Transposed Matrix Vector Product

The formula for matrix vector product $y = Ax$ with an $M \times N$ matrix is:

$$y_i = \sum_{0}^{N-1} a_{ik} b_k$$

The corresponding code to realize this with COO is

```
void matvec(const Vector& x, Vector& y) const {
  for (size_type k = 0; k < arrayData.size(); ++k) {
    y(colIndices[k]) += arrayData[k] * x(rowIndices[k]);
  }
}
```

In some Krylov subspace algorithms it is necessary to compute the transposed matrix product $y = A^T x$. Actually forming the transposed matrix and then multiplying by it can be quite expensive. Instead, we can compute the transposed product according to:

$$y_i = \sum_{0}^{N-1} a_{ki} b_k$$

Deliverable Add a member function `trMatvec()` defined as above as a public member to the COOMatrix class in COO.hpp.

3.7 Tuning (AMATH 583 Only)

Considering the loop-based optimizations we have investigated for matrix-matrix product, discuss (in 200 words or fewer) which of these optimizations might be appropriate for sparse matrix-vector product. Discuss also (in 100 words or fewer) which of the vectorization operations might help to improve performance (and by how much). Include these discussions in a file ps4ex3.txt.

3.8 Written Exercises

Recall the code and your result from Section 2.3. The provided code needed to be modified to obtain a meaningful result for the Maximum Performance denominator in computing efficiency. Once you have successfully obtained a value for your computer, provide the following:

1. Description of what changes you made to the timing code
2. Explanation of why you made those changes
3. Clock rate of your computer
4. Max achieved floating point rate of your timed code
5. (**AMATH 583 only**) Under what circumstances, if any, is your `matvec_student` faster than `matvec_inner` and `matvec_outer`?

Place the above prompts/questions and your responses into a text file `ex4.txt`.

3.9 Extra Credit

Compare the performance of sparse matrix-vector product with no compiler optimizations, -O3, and with -Ofast -march=native. Investigate the assembler produced by the compiler (with -Ofast -march=native) and identify any vectorization that takes place. How much (if any) should these vectorizations help? Although your CPU might not run the resulting binary, you can request that Clang produce code for any architecture (this is known as "cross-compiling"). Compile the sparse matrix code with avx512 options (the following though you can research and add others):

```
-mavx -mavx2 -mxsave -fslp-vectorize-aggressive -mfma -mavx512cd
-mavx512dq -mavx512er -mavx512f -mavx512ifma -mavx512pf -mavx512vbmi -mavx512vl
```

Did any more vectorization (or other optimizations) take place? Include your answers in a file ps4exec.txt.

4 Turning in The Exercises

All your cpp and hpp files, your Makefile, and your txt files should go in the tarball `ps4.tgz`. If necessary, include a text file ref4.txt that includes a list of references (electronic, written, human) if any were used for this assignment.

As with the previous assignment, before you upload the `ps4.tgz` file, it is **very important** that you have confidence in your code passing the automated grading scripts. After testing your code with your own drivers, make use of the python script test_ps4.py by using the command `python test_ps4.py` in your ps4 directory. Once you are convinced your code is exhibiting the correct behavior, upload ps4.tgz to Collect It.

5 Learning Outcomes

At the conclusion of week 3 students will be able to

1. Describe the interface and implementation of the `Matrix` class, including the `operator()()` member function.
2. Implement basic matrix-vector product and matrix-matrix product functions using the external interface of the `Matrix` class.
3. Include or exclude code in a source code file using statements from the `#if` family.
4. Explain the high-level functionality of the `-O3` compiler flag.

5. Derive the mapping from two `Matrix` indices to one `std::vector<double>` index.
6. Write a simple C++ class.
7. Correctly use initializer in the constructor for a C++ class.
8. Write a simple test harness for measuring performance of matrix-vector and matrix-matrix multiply routines.
9. Explain the hardware mechanisms that are leveraged by the hoisting, tiling, blocking, and copy-transpose matrix-matrix optimizations.
10. Explain the difference between column-major and row-major ordering.
11. Derive the (basic) computational complexity of matrix-matrix product and matrix-vector product.

At the conclusion of week 4 students will be able to

1. Name the classes of Intel ISA extensions beginning with MMX and through AVX512.
2. Identify the class of ISA extension associated with “xmm”, “ymm” and “zmm” registers.
3. Identify the vector width associated with “xmm”, “ymm” and “zmm” registers.
4. Define SIMD and MIMD and the original author of those terms.
5. Describe the principles of vectorized math operations and characterize how many floating point operations may be done at one time using vectorized operations.
6. Describe coordinate sparse matrix storage (COO).
7. Derive and describe compressed sparse row storage (CSR).
8. Write a matrix-vector product for COO and CSR.
9. Describe three strategies for diagnosing the optimizations being applied by the Clang compiler.
10. Define “intrinsic” in the context of programming for HPC.