UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 4/24/2018
Problem Set #4 **Due:** 5/1/2018

1 NEW HOMEWORK SUBMISSION FORMAT

All homework assignments from now on will consist two parts:

- **Written Assignment**: The part of the assignment will contain all of the short answer questions and plotting. You will submit a pdf of your responses to the Canvas assignment titled: Written Assignment.
- **Coding Assignment**: This part of the assignment will contain all codes. It will be uploaded to the auto-grader as a tarball like usual.

The purpose of dividing the homework assignment into these two individual parts is to simplify grading. The total amount of points remains the same for each assignment; the only difference is that written answers are now submitted separately from codes.

2 Python Grading Script

We will be using the same python grading script from the last homework. You will be able to download the scripts for each homework assignment from the Canvas page. Make sure to download the tar file for AMATH 483 or for AMATH 583.

IMPORTANT: When you tar your code before submitting, do not tar the candidate folder (or any other folder). You should only tar your code files.

When you download and untar the file you will find the following directories:

- *candidate*: a directory where you should place all your c++ code and makefile.
- *input_files*: a folder storing input files (do not modify)
- *key_output_files*: a folder storing reference output files (do not modify)
- *setting.json*: a file containing information for grading (do not modify)
- *grader.py*: the python grading script (do not modify)

To run the grader, copy your code and your makefile in the candidate directory and execute the command

```
$ python grader.py
```

The output from the grader will look something like this

```
== RUNNING CANDIDATE ===============

1. Loading Settings.json
  PASS
2. Compilation
$ make all
  PASS
3. Copying Input Files to Program
copyied input1.txt
copyied input2.txt
4. Running Programs
```

```
$ ./main_1.exe : EXIT CODE CORRECT
$ ./main_2.exe : FAIL (returned exit code -9, expected 2)
5. Verifying Output Files Exist
file_1.txt: Exists
file_2.txt: Exists
file_3.txt: Exists
6. Comparing Output To Key
file_1.txt : PASS
file_2.txt : PASS
file_3.txt : PASS

== END TEST ========================

Estimated Score: 72/72
```

OUTPUTING SOLUTIONS

Instead of outputting results to the terminal using cout, scripts should now produce text files that contain answers. We will be providing you with all the functions for producing these output files. Inside the files `amath583IO.cpp` and `amath583IO.hpp`, we provide the following functions:

- `void writeNumber(T number, std::string file_path);` [T can be int, float, double]
- `void writeComplexDouble(const std::complex<double> value, std::string file_path)`
- `void writeVector(const Vector &, std::string file_path);`
- `void writeMatrix(const Matrix &, std::string file_path);`
- `void writeCOOMatrix(const COOMatrix &, std::string file_path);`

We will also provide you with the following functions for reading input files

- `double readDouble(const char * file_path);`
- `std::complex<double> readComplexDouble( std::string file_path);`
- `Vector readVector(std::string file_path);`
- `Matrix readMatrix(std::string file_path);`
- `COOMatrix readCOOMatrix(std::string path);`

For example, if you were asked to write a script for adding two vectors and then producing an output file with the result, the code would look something like this:

```
int main (int argc, char *argv[])
{
     Vector x = readVector("input_vector1.txt")
     Vector y = readVector("input_vector2.txt")
     writeVector(x + y, "vec_sum.txt")
}
```

COMPILING C++ CODE

The grader will run the command "make all" to compile your C++ code. This means that you will have to add this additional line to your makefile:

```
all: executable_1.exe executable_2.exe ... executable_n.exe
```

where you should replace executable_1.exe executable_2.exe ... executable_n.exe with a list of all the executable your must produce for the homework. An example makefile for producing main1.exe and main2.exe will look something this

```
all: main1.exe main2.exe

main1.exe: main1.cpp
     c++ main1.cpp -o main1.exe
main2.exe: main2.cpp
     c++ main2.cpp -o main2.exe
```

3 Preliminaries

3.1 The Matrix Code

For this assignment you will be provided with a number of source code files that you should put into a subdirectory `ps4`. The code for the `Matrix` class will be in `Matrix.cpp` and `Matrix.hpp`. The files wil contain most of the examples that we showed in lecture. 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.

For creating performance plots as shown in lecture, there is also a small python script `plot.py`. This program takes a list of names. For each name it looks for a file called name.txt, which should contain the output from running the `bench` program. It will then create a plot in a pdf file that contains performance plots from the data in each file.

For plotting purposes, you will need to pull the new docker image amath583/ps3 that has matplotlib installed. To download this image to your local machine, issue the following command in your terminal:
$ docker pull amath583/ps3

For Mac OS X:

$ docker run -it -v /Users/⟨your-netid⟩/amath583work:/home/amath583/work amath583/ps3

For Windows

$ docker run -it -v C:\Users\⟨your-netid⟩\amath583work:/home/amath583/work amath583/ps3

To plot inside the integrated terminal, you also need to make corresponding changes in the "user settings" of VS Code (check PS1 spec for more details).
Note that the `bench.cpp` program sends its output to `std::cout`. To capture this into a file for use by `plot.py`, you can use the I/O redirection capability in the `bash` shell:

```
$ ./bench 2x2 > 2x2.txt
```

This will send the output of the run with the 2×2 kernel to the file `2x2.txt`. You can then generate a pdf file with plot.py

```
$ python plot.py 2x2
```

Note that no matter what input you give the program it will always create an output `matplot.pdf`. Note also that you can put multiple lines on the same plot

```
$ python plot.py 2x2 4x4 4x2
```

3.2 Matrix Orientation

In lecture we developed the following `Matrix` class:

```
class RowMatrix {
  public:
    RowMatrix(size_t M, size_t N) : num_rows_(M), num_cols_(N), storage_(num_rows_ * num_cols_)

          double& operator()(size_t i, size_t j)       { return storage_[i * num_cols_ + j]; }
    const double& operator()(size_t i, size_t j) const { return storage_[i * num_cols_ + j]; }

    size_t num_rows() const { return num_rows_; }
    size_t num_cols() const { return num_cols_; }

  private:
    size_t              num_rows_, num_cols_;
    std::vector<double> storage_;
  };
```

(We are calling this `RowMatrix` here, for reasons that will be clear as we proceed.) An important consideration in defining a `Matrix` or array class is how to map the two-dimensional indexing of the `Matrix` to the one-dimensional storage mapped to computer memory. In the above example, the `Matrix` is *row-oriented*. That is, the elements in each row are contiguous, and the rows themselves are contiguous in memory.

There is an alternative to having a row-oriented matrix, namely column-oriented. Assume we have a matrix type `ColMatrix`. Since we can overload functions on types, we can define matrix-matrix multiply for `RowMatrix` and `ColMatrix` as follows:

```
void multiply(const RowMatrix& A, const RowMatrix& B, RowMatrix& C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
        C(i, j) += A(i, k) * B(k, j);
      }
    }
  }
}

void multiply(const ColMatrix& A, const ColMatrix& B, ColMatrix& C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
        C(i, j) += A(i, k) * B(k, j);
      }
    }
  }
}
```

We can invoke these as: RowMatrix A(size), B(size), C(size); ColMatrix D(size), E(size), F(size); multiply(A, B, C); multiply(D, E, F); Notice that `RowMatrix` and `ColMatrix` have the same interface – and deliberately so. A matrix has particular semantics and both `RowMatrix` and `ColMatrix` present those. The difference between the two types has nothing to do with how they are used – the difference is simply how they internally represent the matrix.

However, despite having the same interfaces and same semantics, since `RowMatrix` and `ColMatrix` are different types, we have to have different `multiply` functions for them. Looking at the bodies of the functions, the text is *exactly the same* (since the interfaces are exactly the same). It would seem rather a waste of effort to have two functions with exactly the same text that differ only by the types of the arguments.

Moreover, for more complicated functions, this is a maintenance nightmare waiting to happen – when one function is changed, all versions need to be changed. It is much easier to maintain a single function.

This problem is similar to the one we saw with procedural abstraction. That is, when we have the same program text except for differing values, we parameterize the function with those values and use a single function over multiple values. To deal with this issue at a higher level, C++ enables parameterization over types with the **`template`** mechanism. Using **`template`**s, we can write a type-parameterized `multiply` as follows:

```
template <typename MatrixType>
void multiply(const MatrixType& A, const MatrixType& B, Matrixtype& C) {
  for (size_t i = 0; i < A.num_rows(); ++i) {
    for (size_t j = 0; j < B.num_cols(); ++j) {
      for (size_t k = 0; k < A.num_cols(); ++k) {
        C(i, j) += A(i, k) * B(k, j);
      }
    }
  }
}
```

When a function is parameterized this way, it is referred to as a "function template" (not a "template function"). It is parameterized on the **`typename`** `MatrixType`.

We can invoke `multiply` via the function template in the same way as before: RowMatrix A(size), B(size), C(size); ColMatrix D(size), E(size), F(size); multiply(A, B, C); multiply(D, E, F); In fact, we can call the `multiply` function template on any type we like, as long as it provides the interface expected in the body of the function template.

Templates are an extraordinarily powerful mechanism in C++. In fact, the template mechanism in C++ is its own Turing-complete language – one can perform arbitrary compile-time operations using it. (C++ has more recently introduced the **`constexpr`** mechanism for compile-time computation and should be used instead of templates.)

There are a few things to be aware of when using function templates. **Number One.** *Function templates are not functions.* As the name implies, function templates are, well, templates. They are not actual functions and are not compiled – by themselves they don't have enough information to be compiled. The type parameter needs to be bound to an actual type – then the compiler has enough information to compile the function. One refers to the creation of a function when the type parameter is bound as "instantiation." **Number Two.** *The model for using function templates is different than for concrete functions.* To use a concrete function in a program we need its declaration. The function itself can exist elsewhere and be compiled separately from where it is being called. We can't compile a function template separately – we don't even know what it will be until we compose it with a particular type. When the function template is composed with a particular type, *then* the compiler can create an actual function based on this composition (an instantiation) and compile it. Accordingly, we have **Number Three.** *Function templates are included completely in header files.* The full text of the function template needs to be present when it is intantiated, so the full text of the function template is included as its declaration *and* definition in the header file. **Number Four.** *Errors in function templates only become exposed upon composition.* Since the function is compiled only when it is composed with a particular type, we won't see any errors in the function (or with the composition) until the composition occurs. Although error messages are getting better (and C++ will be including methods for better checking of templates), error messages from errors in function templates are notoriously verbose. **Number Five.** *Instantiated function templates perform exactly as the equivalent concrete function.* That is, the function template `multiply` will give exactly the same performance when composed with `RowMatrix` or `RowMatrix` as the earlier functions specifically written for those types.

3.3 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 may 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.)

3.4 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. Note that the data being indexed from x are accessed through an indirection, precluding vectorization in the inner loop. Finally, a question for thought: How much reuse do we (can we) get of A, x, and y?

4 Warm Up

To get some experience with function templates, as well as with matrix multiplication, you will be provided with the following files:

```
Vector.hpp
Vector.cpp
Matrix.hpp
Matrix.cpp
COO.hpp
COO.cpp
amath583.hpp
amath583.cpp
matmat583.hpp
matmat583.cpp
matvec583.hpp
matvec583.cpp
sparsebench.cpp
bench.cpp
amath583IO.hpp
amath583IO.cpp
HRTimer.hpp
Timer.hpp
Makefile
plot.py
```

Concrete functions for `multiply` are contained in matmat583.cpp and the function templates are included in matmat583.hpp. A `RowMatrix` and `ColMatrix` class are included in Matrix.hpp.

4.1 Column Orientation

The `ColMatrix` class is incomplete. Fill in the indexing operators to properly support column orientation. Verify that your implementation correctly implements a matrix.

4.2 Abstraction Penalty

To compare performance between concrete implementations of functions and the equivalent function templates, the template code is selected or not using the preprocessor. If the macro `__TEMPLATED` is defined when the program is compiled, then the function templates will be used, otherwise the concrete functions will be used.

Concrete functions. Compile and run the "bench" program without the macro `__TEMPLATED`. Run this for a variety of sizes and record those times. The program will report times for row oriented and for colum oriented matrices.

Function templates. Execute the command "make clean" to delete the object files and executables in your directory. Enable the definition of the macro `__TEMPLATED` and recompile "bench". Run the program for the same sizes as you did for the concrete functions and compare the times. They should be very close to what you had before.

4.3 Inspecting your CPU

cpuinfo As discussed in lecture, 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/cpuinfo
```

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?

5 Exercises

5.1 Multiple Template Parameters

Just as functions can be parameterized with more than one variable, function templates can be parameterized with more than one type. And, just was with functions, although the return type of a function can't be the only occurence of a type parameter, if the type parameter can be deduced elsewhere, a type parameter can be used as a return type.

Let's look at an example showing type deductions. One of the earliest objects of mathematical study was determining roots of polynomials. One perplexing obstacle came up in the very simple equation: $x^2 = -1$. What is $\sqrt{-1}$? Of course, it turned out that solving polynomials required pairs of numbers rather than single numbers (complex rather than purely real). If we wanted the `sqrt583` function to be able to return complex numbers, we would use an interface such as:

```
template <T>
T sqrt583(double x);
```

This definition is legal. However, to use this function, we have to explicitly specify what we want `T` to be:

```
#include <complex>
template <typename T>
T sqrt583(double x) { /* body of function here */ }
int main(){
  double y = sqrt583<double>(2.0);  // T is double
  std::complex z = sqrt583<complex<std::double> >(-1.0); // T is complex<double>
  double w = sqrt583(3.0); // ERROR, cannot infer T
  return 0;
 }
```

We can use function templates (as we did earlier), without explicitly specifying their type if the compiler can deduce the types from the function input parameters:

```
template <typename T>
T sqrt583(T x) { /* body of function here */ }

int main() {
  double y = sqrt583<double>(2.0);  // T is double
  std::complex z = sqrt583<std::complex<double> >(-1.0); // T is complex<double>
  double w = sqrt583(3.0); // OK, T is inferred as double
  return 0;
}
```

In this case, the final call to `sqrt583` will infer that `T` is a **`double`** and so the function will take a **`double`** and return a **`double`** just as if it had been defined as:

```
  double sqrt583(double x) { /* body of function here */ }
```

This won't automatically solve the problem of a negative real value being passed in. But to handle that case one would invoke:

```
template <typename T>
T sqrt583(T x) { /* body of function here */ }

complex<double> z = sqrt583(std::complex<double> (-1.0, 0.0));
complex<double> w = sqrt583<std::complex<double>> ({ -1.0, 0.0 });
complex<double> w = sqrt583<std::complex<double>> (-1.0);
```

The first expression infers the type of T from the input parameter. The second expression specifies that T is a `complex<double>` and initializes it to the complex number `(-1, 0)`. The final expression also specifies that T is a `complex<double>` but also uses the fact that a double will automatically get promoted to a `complex<double>` if a double is used where a `complex<double>` is required.

5.1.1 Complex Square Root (AMATH583 only)

Extend the square root function that we developed in lecture to operate with complex numbers, as illustrated above. Verify that it will compute the correct answer for negative real numbers as input as well as for complex numbers.

Recall the every number has two square roots. For example the square roots of 4 are ± 2. More generally, if x is the square root of z, then $\tilde{x} = \exp(i\pi)x$ is also a square root. Depending on the initial condition you choose for Newton, you will converge to either of the two square roots. Once you find a valid square root x, you can immediately find the second square root by multiplying x with $exp(i\pi)$ to obtain $\tilde{x} = \exp(i\pi)x$.

The auto-grader will provide your code with a complex number z where $Re(z) \neq 0$ and check to see if your code can correctly find the square root x that lies in the positive real plane (i.e. $R(x) \geq 0$).

Code Deliverable: Write the test program `square_root.exe` that implements the complex square root function. Your program should take one command line argument that corresponds to a file containing a complex double object saved using *writeComplexDouble()*. Check `input_complex_1.txt` in the grader folder for an example of input files. Your program should load the complex double using *readComplexDouble(filename)*, compute the square root that lies in the positive real plane and write the result to the file `square_root.txt` using the function *writeComplexDouble()*.

5.1.2 Matrix Vector Multiply

Write a function template for `matvec` and `operator*` that are parameterized by both the matrix type and the vector type. The prototype is:

```
template <typename MatrixType, typename VectorType>
VectorType operator*(const MatrixType& A, const VectorType& x);
```

Note that you can specify references (and/or const) on type parameters. Put this function in matvec583.hpp (recall that it needs to be in the header). There will be skeletons for these functions in the file `matvec583.hpp`

Code Deliverable: Write the program `mat_vec.exe` that implements matrix vector multiplication. Your program should take two command line arguments. The first corresponds to a file that contains a matrix saved using *writeMatrix()*. The second argument corresponds to a file that contains a vector saved using *writeVector()*. Check `input_matrix_1.txt` and `input_vector_1.txt` in the grader folder for an example of input files. Your program should load the matrix and vector using the functions *readMatrix* and *readVector*, and write the resulting matrix-vector product to the file `mat_vec.txt` using the function *writeVector()*.

5.2 Do Data Access Patterns Matter?

5.2.1 Matrix-Matrix Products

The loop indices in matrix-matrix product are independent of each other. In the function we have been considering, we have been using the "ijk" ordering. However, the loops can be nested in any order of the i, j, and k indices – six versions in all.

Implement ikj and jki. Rename the function template `multiply` to `multiply_ijk`. Using `multiply_ijk` as a pattern, create function templates for `multiply_ikj` and `multiply_jki`, with the loop orderings "ikj" and "jki", respectively. Verify that these functions each correctly compute the same matrix-matrix product for both row-ordered and column-ordered matrices. Hint: Check `matmat583.hpp` for these functions.

Timing ijk, ikj, and jki. Modify "bench.cpp" to time `multiply_ijk`, `multiply_ikj`, and `multiply_jki` for `RowMatrix` and `ColMatrix` (six times in all). Run this for a variety of sizes. There should be some obvious trends among the six runs.

Written Deliverable: Create two plots, one for RowMatrix and one for ColumnMatrix. In each plot show the running time of matrix products for matrices of sizes $2, 4, 8, \ldots, 1024$ using `multiply_ijk`, `multiply_ikj`,

and `multiply_jki`. Using what you know about memory hierarchy, SIMD/vectorization, CPU operation, and data layout for row-major and column-major matrices, explain the behavior you are seeing.

5.2.2 Matrix-Vector Products (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.

In analogy to the previous exercise, create two matrix-vector functions `matvec_ij` and `matvec_ji` in `matvec583.cpp`. These can be function templates or written for the `Matrix` and `Vector` classes we have been using in lecture. The difference between these two functions is that the first should iterate over the vector (over j in the *inner* loop), while the second should iterate over the vector (over j in the *outer* loop). These can be function templates (completely defined in matvec583.hpp) or concrete functions for `RowMatrix` or `Matrix`. If concrete, as per usual, the prototype should go in the header and the implementation in the cpp file (matvec583.hpp and matvec583.cpp, respectively).

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);
```

Compare the runtimes for your ij and ji versions.

Written Deliverable: Make a plot showing the running time of matrix-vector products for matrices and vectors of sizes $2, 4, 8, \ldots, 1024$ using `matvec_ij`, `matvec_ji`. Using what you know about memory hierarchy, SIMD/vectorization, CPU operation, and data layout for row-major and column-major matrices, explain the behavior you are seeing.

5.3 Transposed Sparse Matrix Vector Product

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

$$y_i = \sum_{k=1}^{N} a_{ik} x_k, \qquad i = 1, \ldots, M.$$

The corresponding code to realize this with COO will look something like

```
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]);
  }
}
```

For 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:

$$y_i = \sum_{k=1}^{N} a_{ki} x_k, \qquad i = 1, \ldots, M.$$

Code Deliverable: Add a member function `trMatvec()` defined as above as a public member to the COOMatrix class in COO.hpp. Write the program `smat_T_vec.exe` that implements transpose matrix vector multiplication with a sparse matrix. Your program should take two command line arguments. The

first corresponds to a file that contains a sparse matrix saved using *writeCOOMatrix*. The second argument corresponds to a file that contains a vector saved using *writeVector*. Check `input_coomatrix_1.txt` and `input_vector_1.txt` in the grader folder for an example. Your program should load the matrix and vector using the functions *readCOOMatrix* and *readVector*, and write the resulting matrix-vector product to the file `smat_T_vec.txt` using the function *writeVector*.

6 Turning in The Exercises

All your written responses should be uploaded to canvas in pdf file to the written assignment, and all your .cpp, .hpp files, and Makefile, should go in the tarball `ps4.tgz` and uploaded to the coding assignment. If necessary you should include a list of references (electronic, written, human) in the pdf.

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. Once you are convinced your code is exhibiting the correct behavior, upload ps4.tgz to Collect It.

AMATH 483: Before submitting homework, check that the command "make all" generates the following executables:

1. mat_vec.exe [MatrixFilename] [VectorFilename]
2. smat_T_vec.exe [SparseMatrixFilename] [vectorFilename]

AMATH 583: Before submitting homework, check that the command "make all" generates the following executables:

1. square_root.exe [complexNumberFilename]
2. mat_vec.exe [MatrixFilename] [VectorFilename]
3. smat_T_vec.exe [SparseMatrixFilename] [vectorFilename]

The grader will run each program with the following command(s)

1. `$ square_root.exe input_complex_1.txt`

   which should generate the following output file:

   (a) *square_root.txt*: contains complex double equal to the square root of the the complex double saved in input_complex_1.txt. You should use the functions *readComplexDouble* to read the input file and the function *writeComplexDouble* to produce the output file.

2. `$ mat_vec.exe input_matrix_1.txt input_vector_1.txt`

   which should generate the following output file:

   (a) *mat_vec.txt*: contains a Vector object corresponding to the sum of product of the Matrix saved in input_matrix_1.txt and the vector saved in input_vector_1.txt. You should use the functions *readMatrix* and*readVector* to read in the vectors and the function *writeVector* to produce the output file.

3. `$ smat_T_vec.exe input_sparsematrix_1.txt input_vector_1.txt`

   which should generate the following output file:

   (a) *smat_T_vec.txt*: contains a Vector object corresponding to the product of transpose of the Matrix saved in input_sparsematrix_1.txt and the vector saved in input_vector_1.txt. You should use the functions *readCOOMatrix* and *readVector* to read in the vectors and the function *writeVector* to produce the output file.

7 Learning Outcomes

At the conclusion of week 4 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.
12. Name the classes of Intel ISA extensions beginning with MMX and through AVX512.
13. Identify the class of ISA extension associated with “xmm”, “ymm” and “zmm” registers.
14. Identify the vector width associated with “xmm”, “ymm” and “zmm” registers.
15. Define SIMD and MIMD and the original author of those terms.
16. Describe the principles of vectorized math operations and characterize how many floating point operations may be done at one time using vectorized operations.
17. Describe coordinate sparse matrix storage (COO).
18. Derive and describe compressed sparse row storage (CSR).
19. Write a matrix-vector product for COO and CSR.
20. Describe three strategies for diagnosing the optimizations being applied by the Clang compiler.
21. Define “intrinsic” in the context of programming for HPC.