CS 378 Project 1

Altanali Nagji
Dr. Devangi N. Parikh

January 2023

This is an individual project.

1 Setting up your workflow

This project will give you exposure to the various tools used for HPC software development. You will have the opportunity to design a testsuite, write a driver routine, modify makefiles. This project will also introduce you to the errors associated with floating point computations.
We will be using the generalized-matrix matrix multiplication operation to learn about these various tools.

1.1 Learning outcomes

During the semester, you will learn to appreciate goal-oriented programming, and so we take the same approach to describe the learning outcomes for these projects. For goal-oriented programming we start with the preconditions and postconditions of the program. Preconditions describe the assertions that must be true at the start of the program, and postconditions are the assertions that must be true at the end of the program.

1.2 Preconditions

Before starting this assignment, you should be familiar with

- C language (pointers and memory allocation in particular).
- Setting up your environment so that you can ssh into the CS lab machine and edit code. Ideally, familiarity with editing source code via the command line using vim or emacs. If not, an IDE will do.
- Using preexisting makefiles to build code and run code.
- Running an executable from the command line.
- Cloning a git repository, committing changes, and pulling and pushing to a remote repository. Again, ideally you should know how to do this via the command line.

1.3 Postconditions

- Cloning, building and installing a pre-existing software library.
- Understand the various parts of a makefile and add new and targets as required and make modifications to the recipe by adding compile time flags.
- Understand the layout of matrices within memory, and know how to access various elements of the matrix.
- Understand accuracy of floating point calculations, determine if the results of matrix multiplication is correct and write code to implement the same.

1.4 Setup

You will work on the CS lab machines. If you chose to work on your own machine, we may not be able to help you with your setup.

For this project, and most of the other projects throughout the semester, we will be using BLIS (BLAS-Like Instantiation Software) as our reference code. This means we will be comparing our implementation of matrix multiplication with the high performance implementation provided by BLIS, both in terms of accuracy and performance.

Note: If you copy paste the commands directly on your command line there might be some extra erroneous characters.

To install BLIS:

Cloning the BLIS repository: *Do not clone the BLIS repository in your home directory.* The BLIS source code is hosted on github. To clone BLIS you will run the following on your command line:

```
$ git clone https://github.com/flame/blis.git
$ cd blis
```

Building and installing BLIS: Configure, build and install BLIS with the following commands

```
$ ./configure -p $HOME/blis auto
$ make -j4
$ make check
$ make install
```

The first command configures the BLIS build system. The `-p` option allows the user to specify a common installation path. This will install BLIS in a directory named `blis` in your home directory. The configuration script will automatically detect the architecture of the machine you are working on. Make sure that the hardware detected is `haswell`.

The second command will compile the BLIS source code. The `-j4` option will allow 4 jobs (commands) to run simultaneously.

The third command checks the library has been compiled correctly. Make sure you see the following messages towards the end of the output.

```
All BLIS tests passed!
All BLAS tests passed!
```

The fourth command installs the BLIS library in your `$HOME` directory.

2 Getting the project code

For this project, the code is made available on my github repository. You will duplicate my repository into your own private repository by following these steps:

- On the github web page, create a new private repository. Your repository must be named

  ```
  <eid>-sp23cs378pfcp-project1
  ```

  Make sure the repository is private.

- On your favorite terminal, create a bare clone of my repository

  ```
  $ git clone --bare https://github.com/dnparikh/sp23cs378pfcp-project1.git
  ```

- Mirror-push to your private repository.

  ```
  $ cd sp23cs378pfcp-project1.git
  $ git push --mirror https://github.com/<username>/<eid>-sp23cs378pfcp-project1.git
  ```

- Remove the temporary local repository you created earlier.

  ```
  $ cd ..
  $ rm -rf sp23cs378pfcp-project1.git
  ```

- You should be able to see the starter code in your private repository.

- Now, you must clone your private repository so you can get started.

  ```
  $ git clone git@github.com:<username>/<eid>-sp23cs378pfcp-project1.git
  ```

- `cd` into the your repository on your command line.

```
$ cd <eid>-sp23cs378pfcp-project1
```

- Build and run the starter code

```
make IJP
```

  This should compile and run your code. Make sure the code runs properly. If you should get an error, read the error, and figure out how to fix it. The error most likely is because of your set up. Understand the errors and fix them accordingly.

Before you start modifying the code, read through the code and the makefile and at a high level understand what the code is doing.

`driver.c` This is the file where `main()` is written. `main()` scans the matrix sizes to be tested from the command line at runtime. It sets up the matrices, and fills them with randomized double precision floating point numbers. The code calls to the BLAS API `dgemm_` as the reference code, and times this call. It also calls `MyGemm`, which at the moment is a triple-nested loop implementation of `dgemm()`. The code then compares the entries of the resulting C from the reference implementation and `MyGemm()`, and reports the maximum absolute difference of the two implementations. We expect this difference to be lower than 10^{-12}.

Things you should look at and understand:

- what each of the columns that gets printed out represents
- makefile–what are the various variables used. What make targets are available, what is the corresponding recipes.
- what happens when you run `make IJP`.

In the current code

- the parameter of leading dimension `ld<matrixname>` is used to stride through the matrix
- matrix sizes to be tested are scanned from the command line at run time
- Calls the BLAS API to `dgemm_` as the reference code.

3 Things you will implement

There are various tiers of things you will implement for this project. You will be grading based on the features you chose to implement, and how you implement them.
The files you may have to edit for a particular task will be listed in ⟨brackets⟩.

3.1 Beginning

Row and column stride: Modify the code so that you are using the row stride (`rs<matrixname>`) and column stride (`cs<matrixname>`) to stride through the matrix when you are computing the matrix multiplication This change will have to be made in `Gemm_IJP.c`. This will also require a modification of all functions that are called in the driver that traverse the matrices. This will require changes to be made in `driver.c, MaxAbsDiff.c, RandomMatrix.c`, Your matrices should be stored in column major order. If you implement this in a destructive way (original code does not remain part of the code you turn in) it will be considered a beginning level.

Your GEMM API must be of the following following:

```
MyGemm( int m, int n, int k,
        double *A, int rsA, int csA,
        double *B, int rsB, int csB,
        double *C, int rsC, int csC )
```

Using the BLIS-typed API: This change goes in tandem with the above change. Since we are now using row and column strides to traverse the matrix, for the reference implementation we will have to use the BLIS-typed API. This will require you to make changes in `driver.c` You can find documentation for this API here. You will find sample sample code that uses this api here.

3.2 Developing

Get matrix sizes at compile time: Instead of using `scanf()` to obtain the matrix sizes using preprocesser flags that are set at compile time through the the makefile. Changes will have to made in `driver.c` and `Makefile`. I have provided some hints both in `driver.c` and `Makefile`.

Debug flags: Create a debug flag `DEBUG` that can be set using that runs just one matrix size. Changes will have to made in `Makefile` and possibly `driver.c` depending on how you chose to do this.

You can read more about preprocesser flags in the GCC manual under the heading "Options Controlling the Preprocessor". You can find GCC manual here. You can the GCC documentation on macros in C here and useful information on "conditional directives" used by the preprocessor here.

3.3 Advanced

Different problem sizes: Use a preprocesser time flag to switch different problem size ranges. Test two different ranges of problem. Create different makefile targets to build and run these. For this changes will have to be made in the `Makefile`

Column-Major/Row-major stored matrices: Use preprocessor flags that are set in the makefile to switch between row major/column major stored matrices to run. This will require changes to be made to

`driver.c` and `Makefile`.

3.4 Exemplary

Switch between ld/(rs/cs): Use preprocessor flags to switch between using the leading dimension and rs/cs to stride the matrices. The possible places that changes will have to be made are in `driver.c`, `Gemm_IJP.c`, `MaxAbsDiff.c`, `RandomMatrix.c`, and `Makefile`.

Debugging: Create debug flags that reports the first index of where the computation went wrong and early exists. Changes will have to made in `MaxAbsDiff.c` and and `Makefile`.

Errors: Write a routine that reports the maximum element-wise relative error. The relative error between two elements is given by

$$\text{relative error} = \frac{|\gamma - \gamma_{ref}|}{|\gamma_{ref}|}$$

Implement this in `MaxAbsDiff.c`

4 Submission

TBD

Give us a few days to test and write up the instructions.