PA1 Instructions

- **checkpoint due: 2022/4/10 11:00 pm**
- **report due: 2022/4/21 11:00 pm**

In assignment #1, you or you and your partner will use your knowledge of the memory hierarchy and vectorization to optimize a matrix multiply routine. We will provide you with simple starter code. Your job is to optimize that starter code to achieve higher performance. Your multiply routine will be compared to OpenBLAS - http://openblas.net [Blas is an optimized basic linear algebra subroutine]. You will optimize your code for maximum performance on one core of an Amazon EC2 t2.micro instance. We don't expect you to achieve the same performance as OpenBLAS - see performance goals below.

The assignment has a programming component and an analysis component.

WARNING: this is a deceptively short program that will really challenge your optimization skills. START EARLY - it will take more effort than you think. These instructions are long, but it's important to read and understand them.

Part 1. Getting the starter code and GIT repository

All assignments will be turned in using classroom.github.com. The link to the github classroom is:. https://classroom.github.com/a/j-B0jQwx (READ THE FOLLOWING PARAGRAPH BEFORE CLICKING THIS LINK).

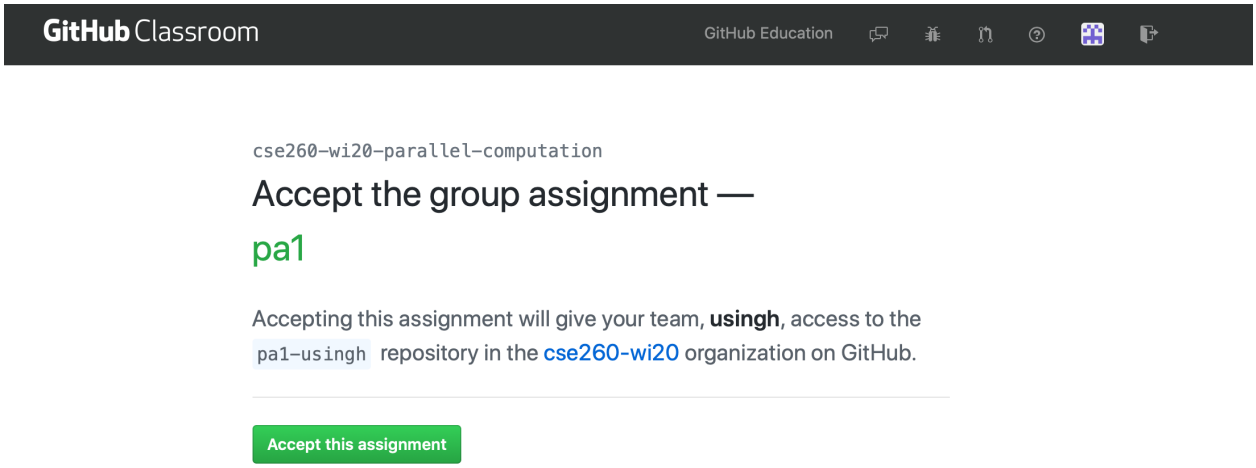
You should be working in a team of 2 for the assignment. The git link will take you to a page where you can either create a new team or join an existing team. One of the team members will create the team and the other will join the team. Please use the following naming convention for your team name (this will name your repository appropriately so our autograding scripts will work correctly).

pa1-userID-userID

Use your **UCSD user ID (email ID)**, for example, tsundara or b5chin. The user ids should be listed in alphabetical order. For example, hw1-b5chin-tsundara and NOT hw1-tsundara-b5chin. It's ok if classroom tacks on an extra pa1 so you might end up with hw1-hw1-b5chin-tsundara.

This web site - https://education.github.community/t/how-to-create-teams-in-github-classroom/9394/2 describes the process of creating and joining a team.

The team selection or create page will look like this:



Setting up AWS Instance

We need an Ubuntu instance for this PA. Please check the course website for how to create such an instance. After you created your AWS instance and connected to it, you need to add several packages. In command line, type:

```
sudo apt-get update
sudo apt install make
sudo apt install g++
sudo apt install libopenblas-dev
```

Important: You have to submit a checkpoint with your partner's name & signature in the template provided to you and submit it on Gradescope by Sunday i.e. April 10, 11:00PM.

Part 2. The Assignment Details

Your grade will be determined by your achieved performance (post your performance results on piazza to challenge your classmates or get a sense of how you are doing versus the rest of the class); correctness on our test set; well-displayed results and comments/exposition/insight on how you got your results and why the program behaves as it does. The write-up of explanations and insights has higher weight than the program unless your program is essentially the naive code or our starter blocked code (see grading below).

1. We provided you with code based on the BLISlab tutorial: https://github.com/flame/blislab

   We have modified the blislab tutorial in the following ways.

   - our input and output matrices are stored in row-major order
   - our implementation must take care of cases where *n* is not a multiple of the various blocking factors in the code. That is, our program can handle any values of *n* (for square matrices).

2. Familiarize yourself with matrix multiplication by looking at the naive implementation we have provided in the starter code. Run the code and establish a baseline performance for naive multiplication.

3. Read the BLISLab tutorial. Rather than have you do the entire tutorial, we focussed on *Step3 - (Section 4 - Blocking for Multiple Levels of Cache)*. However, please read the tutorial paying particular attention to earlier sections: *2.4.1 - Using Pointers*, *2.4.2 - Loop Unrolling*, *2.4.3 - Register Variables*, *3.1-3.3 Poorman's BLAS*. *Sections 3.1-3.3* describe the general technique for blocking. Blocking amortizes the cost of moving data from main memory to higher speed memory (caches, registers) over the cost of computation. By decomposing the larger matrix into smaller sub-matrix problems, Poorman's BLAS (*blislab tutorial 3.2*) may achieve similar performance for larger matrices that do not fit in the CPU caches and smaller matrices that do fit in the CPU caches.

4. Read the *BLISLab tutorial Section 4 (Step 3)* very carefully. **You will have to read it several times**. Recall that we have made two changes to the tutorial
   a. implement row-major matrices for input matrices A, B and the output matrix C
   b. handle fringe cases where N is not an even multiple of the various blocking sizes.
5. After having read *BLISlab tutorial section 4,* read and understand this description of our starter code. Our description is different from the BLISlab tutorial (you should answer why you think this might be the case in your analysis). Here is a simplified snippet based on the starter code with an analysis. Again, note the differences between this code and the BLISlab tutorial.

```
Loop 5: for ic = 0; ic < m; ic += Mc
          partition C into panels Cj of Mc x n
          partition A into blocks of Mc x n
Loop 4:   for pc = 0; pc < k; pc += Kc
            partition A (Mc x n) into panels of Mc x Kc (Ap)
            pack Ap into subpanels Mr x Kc
            partition B into panels of Kc x n (Bp)
Loop 3:     for jc = 0; jc < n; jc += Nc
              pack Bp into subpanels Kc x Nr
              partition Cj into panels of Mc x Nc
                // macrokernel
```

```
Loop 2:          for ir = 0; ir < Mc; ic += Mr
Loop 1:            for jr =0; jr < Nc; jr += Nr
Loop 0:              for kr=0; kr < Kc; kr++
                         // microkernel
                         C_Mr x Nr += A_Mr x Kc x B_Kc x Nr
```

Loop 5 breaks the problem into strips that are Mc high. This is shown in Figure 1. Note C is broken into strips called Cj and A into strips of Mc x n.

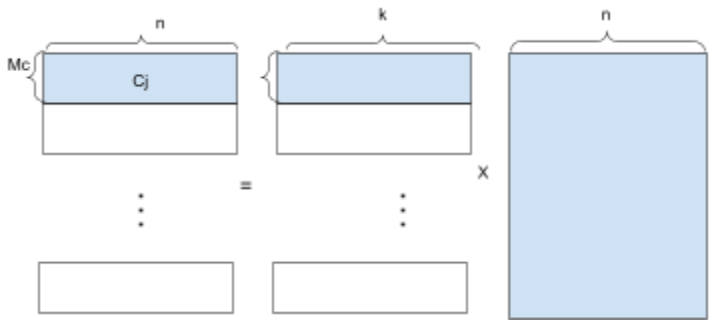


Figure 1.

In Loop 4, a strip of A from loop 5 (Mc x k) is partitioned into panels Ap that are rank Mc x Kc and B is partitioned into panels Bp of rank Kc x n (Figure 2).

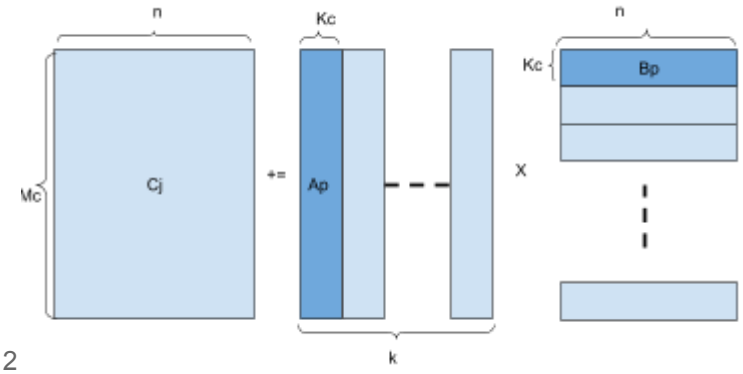


Figure 2

Within loop 4, Ap is then packed into subpanels of rank Mr x Kc. (Figure 3). The subpanels are packed in a special way such that the microkernel will be able to access elements it needs in sequential address order (i.e. in a cache friendly way). The highlighted region represents one Ap. Each subpanel is shown with a zig-zag line. The zig-zag line indicates increasing memory address so that elements down a column of Mr are sequentially addressed in memory. After a subpanel is packed, the next subpanel is packed until the entire Ap has been packed.

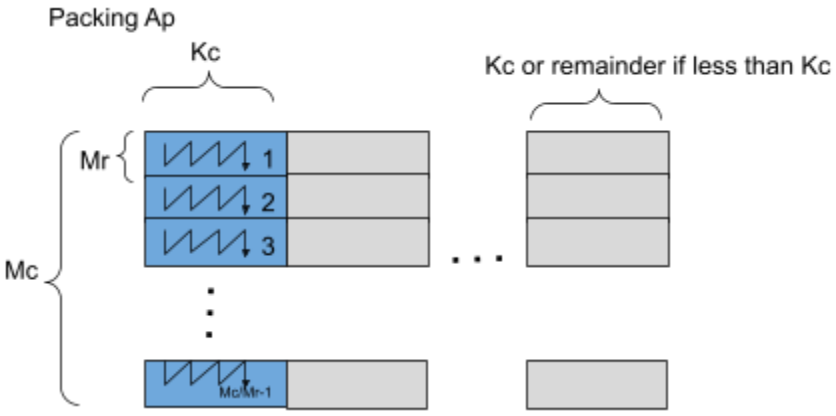


Figure 3.

Loop 3 divides Cj and Bp into columns that are Nc wide (Bj) (Figure 4) and then packs each Bj into subpanels that are rank Kc x Nr. (Figure 5). Similar to Ap, the small subpanels of Kc x Nr are packed as illustrated by the zig-zag line.

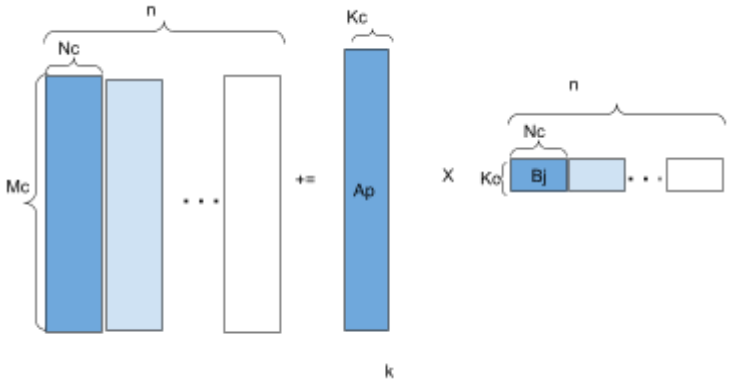


Figure 4.

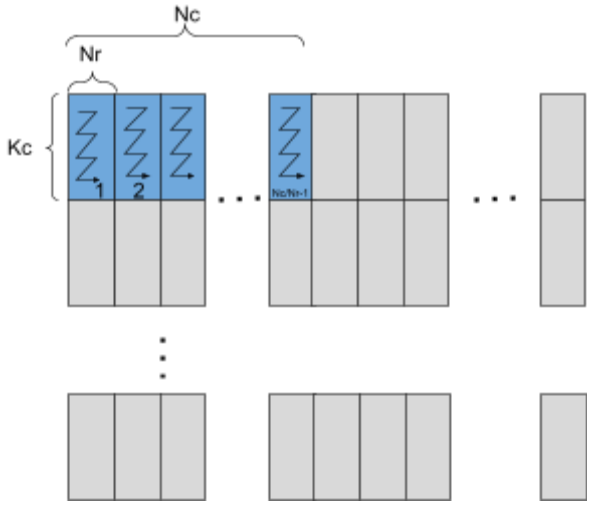


Figure 5. Packing Bj

Loops 2 and 1 comprise the macro kernel. This kernel will multiply the packed Ap by the packed Bp. Loop 2 (Figure 6) iterates over rows (i) and loop 1 (Figure 7) iterates over columns (j). Since the inner loop iterates over columns while keeping the row constant, the subpanel of Ap is kept in the L1 cache, while successive column subpanels of Bp are moved in and out of the L1 from the L2 [2].

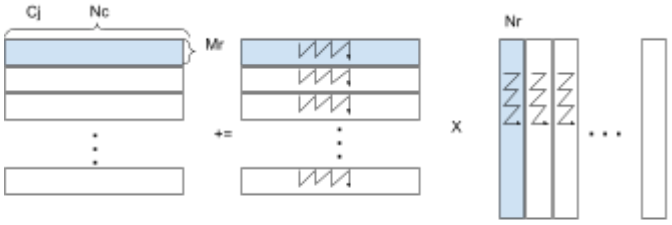


Figure 6.

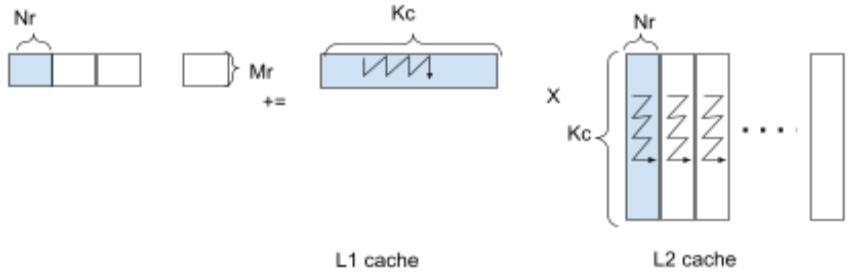


Figure 7.

Finally the micro-kernel performs a matrix multiplication of the two subpanels of A and B to calculate a rank Mr x Nr sub-matrix of C. (Figure 8).

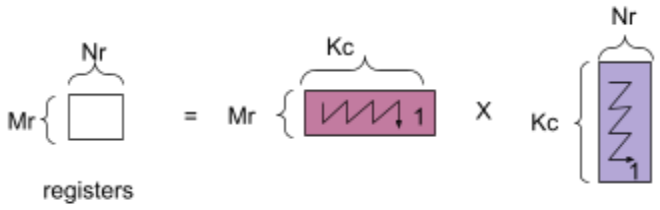


Figure 8

Figure 9 shows the entire process (modified from [1]).

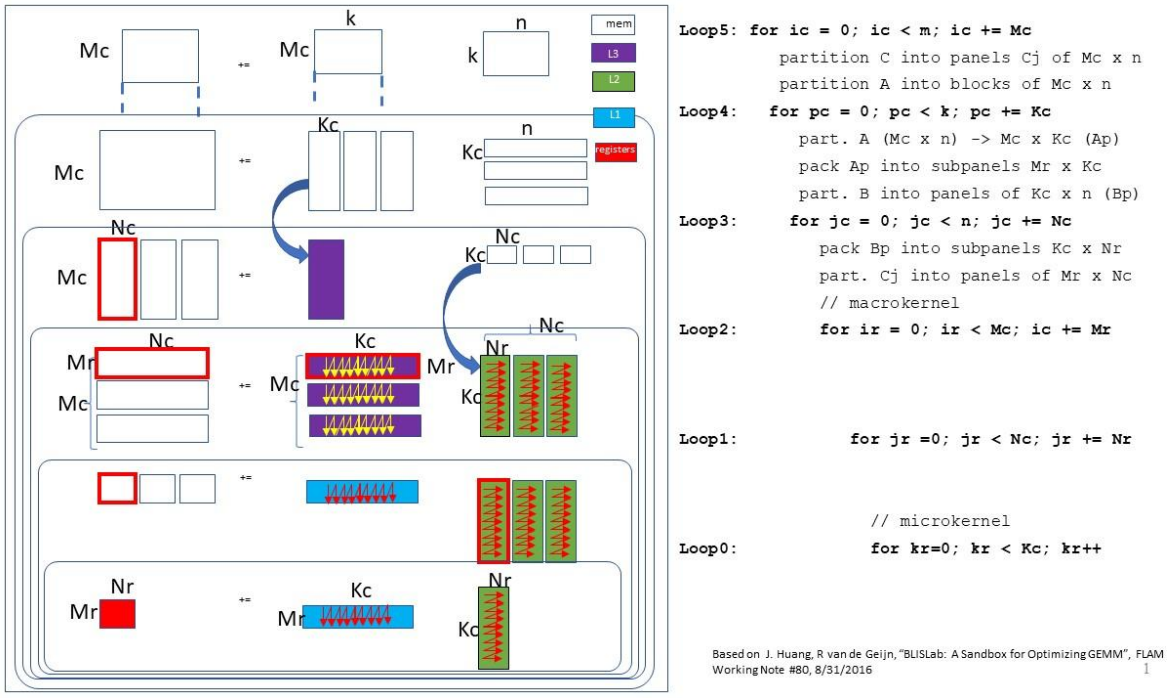


Figure 9.

6. Start by writing packing routines that implement the special packing described above. Get your program to work with your packing routine and the simple C based microkernel we have provided bl_dgemm_ukr.c

7. Verify correctness. The main driver program contains a test harness which will check for correctness. Make sure your code works with the unmodified benchmark.c. When we

test your program we will be using the stock version of benchmark.c. Benchmark.c makes sure that your matrix results match the expected values by less than epsilon (why do you think we do this instead of looking for an exact match?). There are several utilities we have provided to help you debug. You can use the "-i" option to show a small matrix multiplication of an identity matrix multiplied by a matrix with sequential values (see **identMat()**, **seqMat()** and **printMat()**).

8. Implement an optimized AVX2 based microkernel and debug.
9. Tune your parameters, DGEMM_NR, DGEMM_MR, DGEMM_MC, DGEMM_NC, DGEMM_KC.
10. Other optimizations to improve performance. Vectorize the inner loop. T2.micro has AVX, AVX2 and FMA instruction extensions. You enable these with -mavx -mavx2 and -mfma compiler options [see tips below in "Optimization and Development Tips" for hints on how to optimize performance]. Your goal should be 55-65% the performance of OpenBLAS. Running benchmark-blas will give you OpenBLAS performance numbers. The blislab tutorial shows two different ways to use SIMD instructions, one method using broadcast load operations, and the other using permute and shuffle instructions. You may want to try both methods.
11. Your program should
    - use the O(n^3 kernel), not the Strassen kernel or other algorithm which reduces the number of multiply adds
    - use double precision (64-bit) arithmetic
    - handle values of n including non-power of 2.
    - you will change bl_dgemm_ukr.c (to vectorize the innermost loop) and my_dgemm.c (packing and handling of values of n that are not powers of 2). If you want to use c++ please talk to us first to make sure your code compiles with our makefile.

Starter Code

When you checkout your repository, there will be some starter code. The starter code includes:

- Makefile
- benchmark.c - main driver program
- dgemm-blas.c - call's the OpenBLAS library
- my_dgemm.c - this has the skeleton code for blislab.
- bl_dgemm_ukr.c - this has the skeleton code for blislab.
- dgemm-naive.c - simple ijk version of matrix multiply.

You can change the optimization settings in Makefile by changing the MY_OPT. More details in "what to submit" section.

Taking Measurements

The provided code uses gettimeofday() to measure the wall clock time (execution time) and calculate GFlops/sec. For any given N and executable, make sure you take multiple readings (perhaps 3) and if they are reasonably close, use the highest performance number.

You may want to write a python, lua, matlab or perl [or whatever scripting language you like] script to run the program and take the results and plot them (see matplotlib.py or gnuplot) - but you can also just use excel and cut and paste.

Developing Code

You can develop your code pretty much anywhere, but keep in mind that different CPUs will have different cache characteristics so be sure to do some measurements on ec2. [When I was experimenting with this assignment, I did my development on a Windows 10 machine running Ubuntu in WSL2. I moved the code periodically to ec2 to see how I was doing on the target performance goals]. It's easy to move your code if you use the git repository to transfer your files (add / commit / push and pull). Make sure that when measuring performance, that your EC2 instance is set to unlimited so it is not throttled.

To develop (edit, compile, run) code on AWS you are free to use whatever method is most convenient to you. However, there are a few tips on the course website in "Others-> Developing on AWS".

Optimization and Development Tips

The first step is to implement the basic packing routines as described above. You'll need to implement 3 level blocking. The blocking might need to consider cache size, and associativity. There are many factors affecting performance and educated trial and error is a reasonable approach. Modify the starter code my_dgemm.c to implement and correctly call your packing routine (undefine "NOPACK"). Modify bl_dgemm_ukr.c 's basic microkernel to use the packed matrices correctly (hint: you will need to use different leading indices as in the default kernel we do not use the packed A and B submatrices).

Next, you might want to create register tiles for the microkernel while still implementing the microkernel in C. For example, try to make a C based microkernel that does 4x4 multiplication. This technique treats the register file as the fastest memory and tries to make maximal use of values in registers (just like cache blocking tries to make maximal use of values in cache). You can give the compiler hints in C by using the register keyword when defining variables. (Another interesting keyword is restrict https://en.wikipedia.org/wiki/Restrict which may help your performance).

You will then want to implement some fixed sized microkernels based on SIMD using AVX2 intrinsics. On t2.micro, there are 16x256 bit AVX registers. The register tiling is related to AVX vectorization because each AVX register is 256-bits wide (and therefore gives you 4 double precision numbers - that is a vector length of 4).

The design space is wide and you will probably want to think about what microkernel geometries work best and implement a few different ones using SIMD intrinsics. You will want to tune the parameters DGEMM_MR, DGEMM_NR, DGEMM_MC, DGEMM_NC, DGEMM_KC. ([2]) or you can try the autotuning approach([10]). That is, write a perl or python script that compiles and runs your programs with different options to explore the search space. You'll want to consider non-square tile sizes that optimize for maximal use of cache line size or page size.

You may also want to manually unroll the inner loop and see what effect this has on performance.

You may look at code from libraries to get some inspiration on things to try (e.g. BLISLab tutorial code, GotoBLAS or ATLAS. DONOT COPY SOURCE CODE. DOING THIS WILL RESULT IN AN AUTOMATIC ACADEMIC INTEGRITY VIOLATION) - beware this is a slippery slope (ACADEMIC INTEGRITY)

DO NOT implement Strassen's algorithm to speed-up the inner kernel. This reduces the number of multiply-adds which will skew your results.

Part 3. Grading Specifics

3 pts - Checkpoint (team name, EC2 instance name)

2 pts - Correctly Filled out Team Eval - teameval.txt (even for teams of 1)

DECLARATION.TXT (signed and dated by up to two team members)

35 pts - Performance - we will choose reasonable sizes on which to test your performance.

On EC2 t2.micro, we are seeing about 31 to 34 Gf/sec for the OpenBLAS code (N > 256).

The performance points will be awarded roughly according to the following for the larger matrix sizes (Average for N > 512)

>=GF	pts
20	35

18	**32.5**
14	**28 (1/2 performance point)**

We will refine this scale as more data (from your experiments comes in, but we don't expect the topend to change).

Correctness - 10 pts

Results - 15 pts

Analysis - 33 pts

References - 2pts

Part 4. Turning it in

Checking in

You will check everything into classroom.github.com (your git repository). **You should be checking in periodically during your development**. This has several benefits.

1) provide a nice way to support academic integrity by showing your progress and work over time.

2) it provides us with an audit trail to check your development progress and potentially resolve code issues when running your final code.

3) protect you from data loss

4) consistent way to port your code between machines (e.g. your local machine and amazon).

These are the steps for the final submission.

1. run *git status* to see what files have changed.
2. use *git add* to stage your changes for commit.
3. use *git commit -m "FINAL SUBMISSION"* to commit your staged files into your local repository. The -m FINAL SUBMISSION tag lets us know that you intend this to be your final submission. If you submit again before the deadline, we will also consider the last submission before the deadline as your final (but please also tag that with FINAL SUBMISSION message).

4. push your changes into github with *git push origin main.* For those more familiar with git, you may also be merging branches at this point. **We want the final code to be on the main branch. Don't forget this step. IF you don't push your changes, we won't see them.**
5. Surf over to your repository or clone a new copy to see if your changes are really there.

Things to watch out for:

- Watch out for (and resolve) merge conflicts before your final push!!

If you and your partner have modified files on multiple computers, you may have merge issues. This happens because your code from one repository is out of sync with your code from another and git doesn't know who to merge the files. MERGE CONFLICTS will prevent your code from compiling so be sure to resolve these before you do your final push.

Don't let these problems bite you in the end. If you add/commit/push often, this is less likely to happen.

- As a team, push just the team repository. You shouldn't have any others.
- Be careful with the 'FINAL SUBMISSION' message. You shouldn't have multiple repositories (but if you do somehow), make sure only the one you want graded has this message
- You can submit multiple FINAL SUBMISSIONS before the deadline. We will grade the latest one before the deadline.
- Late work will be accepted as per Grading guidelines on Grading
- DO NOT commit binaries or .o files - just source code. PDF, XLS and txt files are okay.

What to submit

source code

We will use our framework with your blislab/my_dgemm.c and blislab/bl_dgemm_ukr.c. But do include all of your source files (including your Makefile) in case we have trouble compiling, we can refer to your environment. Include a file called OPTIONS.TXT if you use other than the default Makefile options (no need to include if you don't have additional options). In this file, include the line MY_OPT=*whatever your options are*

We expect to be able to do the following

make "`cat OPTIONS.TXT`" benchmark-blislab

to make your program.

Two text documents

You must include the two following documents (a penalty will be accessed if forms are missing or not filled out properly). We will not report a grade without these forms.

1. teameval.txt - completed self-evaluation discussing your work. If in a team, this form addresses the division of labor and other aspects of how the team worked or didn't work.
2. A file named DECLARATION.TXT. This was included in the starter code. Just sign (type) your name(s) and dates to acknowledge compliance with the academic integrity policy. **YOU MUST HAVE FILLED OUT A DECLARATION.TXT OR WE WILL NOT GRADE YOUR ASSIGNMENT AND IT MAY BE CONSIDERED LATE or UNSUBMITTED.**

data file

Generate data.txt using the genDATA.sh script provided in the repository. The Report template states: “Give a performance study for a few values(about 20 different values) of N from 32 to 2048 on your optimized code - data should be in the file data.txt”

There should not be any other text in this file.

Lab Report

Name this file according to your repository name. For example PA1-jdoe-jbeeb.pdf. **YOU MUST USE THE TEMPLATE PROVIDED, HOWEVER, YOU MAY EXPAND THE SPACE FOR THE ANSWERS. ON GRADESCOPE, IDENTIFY THE REGIONS CORRESPONDING TO EACH QUESTION.**

IMPORTANT : check this file into the repository AND **also submit it to gradescope - If you are a member of a team, check the report in once and make gradescope aware of your team members**. It is very important that the file be loaded to gradescope. We will not grade the report unless it is uploaded to gradescope. If you have a partner, upload the file once and associate the file with both team members on gradescope (come see us if you are unsure how to do this).

EXTRA CREDIT:

Do not attempt extra credit until you have completed the base requirements. No extra credit points will be awarded for extra credit if the base assignment is not minimally complete. We will release more detail about it in the near future.

Part 5. Sources/References

Here are some places to look for help (besides coming to us).

Primary Sources

[1] **https://github.com/flame/blislab** - Jianyu Huang, Robert A. van de Guijn, BLISlab: A Sandbox for Optimizing GEMM, August 31, 2016
[2] T Low, F Igual, T Smith, E Quintana-Orti, “Analytical Modeling is Enough for High-Performance BLIS”, ACM Transactions On Mathematical Software, Vol 43, No. 2, August 2016
[3] Jim Demmel On "Desiging fast linear algebra kernels in the presence of memory hierarchies" Jim Demmel's on-line reader on BLAS kernels. Look at Algorithm 3: Square blocked matrix multiply. (useful to get the basic ideas of blocking)
[4] Jim Demmel Lecture slides
http://www.cs.berkeley.edu/~demmel/cs267_Spr12/Lectures/lecture02_memhier_jwd12.ppt
[5] Brian Van Straalen's notes on vectorizing with SSE intrinsics
[6] Interactive Intel SIMD reference pages
https://software.intel.com/sites/landingpage/IntrinsicsGuide/
[7] K. Goto, R, Geijn, "Anatomy of high-performance matrix multiplication", ACM Transactions on Mathematical Software, May 2008 - pretty exhaustive (and a bit hard to follow) description of various tiling strategies. https://dl.acm.org/citation.cfm?id=1356053
[8] Markus Puschel, "How to Write Fast Numerical Code", **https://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring15/slides/10-simd.pdf** describes vectorization with SSE (paper Chellappa, Franchetti, Puschel, "How to write Fast Numerical Code: A Small Introduction"
https://link.springer.com/content/pdf/10.1007%2F978-3-540-88643-3_5.pdf
[9] Bilmes, et al. http://www1.icsi.berkeley.edu/~bilmes/phipac/ - PHiPAC is a code-generating auto-tuner that started as a homework for CS267 at Berkeley. Also see [10] ATLAS
http://math-atlas.sourceforge.net/
[11] Lam et al on, "The Cache Performance and Optimizations of Blocked Algorithms", https://suif.stanford.edu/papers/lam-asplos91.pdf - seminal work on cache blocking

Documentation

[12] Very useful Intrinsics interactive table
https://software.intel.com/sites/landingpage/IntrinsicsGuide
[13] GCC https://gcc.gnu.org/onlinedocs/
[14] Overview of AVX:
https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions
[15] Software Optimization Resources, Agner Fog http://agner.org/optimize/
[16] Koharchik and Jones, "An Introduction to GCC Compiler Intrinsics in Vector Processing", Linux Journal
http://www.linuxjournal.com/content/introduction-gcc-compiler-intrinsics-vector-processing?page=0,0