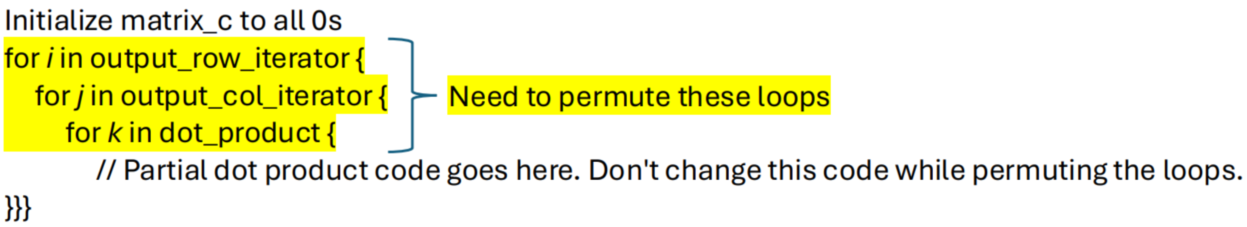
Assignment 0

Introduction:

In this assignment, you are required to implement sequential matrix multiplication using C/C++. Produce *C=AB*, where *A* and *B* are the input matrices. This sequential algorithm makes use of triple nested loops. Let us label these loops so:

1. *i* for row iterator of output matrix *C*
2. *j* for column iterator of output matrix *C*
3. *k* for taking the dot product of i^{th} row of matrix *A* and the j^{th} column of matrix *B*.

Here is the basic code skeleton:



```
Initialize matrix_c to all 0s
for i in output_row_iterator {
   for j in output_col_iterator {
      for k in dot_product {
         // Partial dot product code goes here. Don't change this code while permuting the loops.
}}}
```

Need to permute these loops

Extra variables for storing the intermediate dot product result *should not* be used. The code should read the data directly from the input matrices and store the half-baked intermediate result into the output matrix. Please follow this suggestion to get the proper observations.

Your code should be kept modular. Make sure that you are coding the variants of this triple nested loops in a set of separate functions named "matrixMultiplyXYZ". Here, XYZ is the order of the nesting. For example, one of the functions is matrixMultiplyIJK. You will have six such functions (see below). You can decide the parameters to pass to these functions. Your "main" function should offload the matrix multiplication task to the appropriate "matrixMultiplyXYZ" function depending on the command line argument. Perform two experiments.

Experiment 1:

Execute the above-mentioned variants of matrix multiplication code (all the possible loop permutations -- ijk, ikj, kij, ... etc.) after compiling your code with "–O3" and observe the time reported by the driver (see below). Measure the performance of your matrix multiplication on matrices of sizes 1000x1000, 2000x2000, .. 8000x8000.

Note that "O3" is specific to GCC and LLVM compilers. "O3" enables a slew of compiler optimizations. Optimization flags for GCC can be found at https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. In particular, see -ftreevectorize, -fopt-info-vec-optimized, and –mavx512f. Your submissions should run with g++ version 11.2 with std c++17. (You may also try icpx.)

Note that:

1. There are 6 loop permutations
2. All these permutations should give the same result matrix *C*. Their execution times may vary.
3. Don't change the inner partial dot product code for each of the loop permutation.

For an analysis of this experiment, see https://medium.com/@Styp/why-loops-do-matter-a-story-of-matrix-multiplication-cache-access-patterns-and-java-859111845c25

Experiment 2:

Profile your codes' performance using perf and gprof on data of sizes 1Kx1K .. 8Kx8K.

Linux perf command:

Linux perf provides access to the built-in performance counters on modern CPUs. For example, it can be used to fetch the number of cache accesses and misses on executing a given command (program). You will gather values and plot performance graphs for this assignment. (Official documentation of perf command is here: https://perfwiki.github.io/main/. A nice introduction can be found here: https://www.baeldung.com/linux/analyze-cache-misses.)

You will measure the performance of your matrix multiplication on matrices of different sizes. For each experiment, report the total number of cache misses and cache-hit rate (as a percent of total cache accesses), the CPU time taken by your execution, the CPU time taken by each matrixMultiplyXYZ routine, the percent of the CPU time taken by matrixMultiplyXYZ, readMatrix, and writeMatrix functions (see below). Each should be graphed in a separate curve to be included in your report. Raw data should also be provided in the format listed later.

GNU profiler (gprof) tool:

The gprof tool is used to perform the performance analysis of the application that is compiled using the GNU toolchain (GCC, G++, ... etc.). A good tutorial can be found here: https://hpc-wiki.info/hpc/Gprof_Tutorial. Using this tool, you can find the time elapsed for executing each and every function that is part of your code (and find the most expensive parts of your program). You are encouraged to compare perf numbers and gprof numbers.

Correctness:

A python driver is provided that will use your Makefile and test your code. It generates input matrices, calls your program (see the command line parameters), and verifies your product matrix. It will help you understand the running of your executable.

Your code should be OK if it displays "Test Case passed" in all the successive executions. (Your code may fail when numpy employs internal numerical stability algorithm while calculating the matrix multiplication. In such a case, increasing the tolerance to 1e-5 in the "allclose" function is acceptable.)

Please do change the actual parameters passed to the function "execute_test_case" for differing matrix sizes 1000x1000, 2000x2000 .. 8000x8000.

Submission Rules 1 (If the driver fails to run, you will earn 0):

Following steps must be performed in your code so that it integrates well with the python script:

1. The name of your binary executable file should be "main" (please refer the python script for the execution command)
2. The "main" binary executable should take the following command line arguments: <type> <mtx_A_rows> <mtx_A_cols> <mtx_B_cols> <input_path> <output_path>
    a. type = 0-5. 0 = IJK, 1 = IKJ, 2=JIK, 3=JKI, 4=KIJ, 5=KJI
    b. mtx_A_rows = number of rows in matrix *A*
    c. mtx_A_cols = number of columns in matrix *A* = number of rows in matrix *B*
    d. mtx_B_cols = number of columns in matrix *B*
    e. input_path = path where you can find both the input matrices binary file (mtx_A.bin and mtx_B.bin)
    f. output_path = path where you must store the result matrix *C* in mtx_C.bin file in binary format
3. Write a function "readMatrix" to read a binary row-major ordered matrix of double values (8 bytes per double in little-endian order) from a binary file. Use readMatrix to read matrices from mtx_A.bin and mtx_B.bin.
4. Perform matrix multiplication using the selected triple-nested loop algorithm and store the result matrix in "double" array type.

5. The result array should then be written in mtx_C.bin file in binary format row-major ordering. Please create a function named “writeMatrix” for writing this result matrix into the .bin file.

Note that the input matrices .bin files, that will be created by the python script, should be consumed by your code. Your result matrix, that should be saved in mtx_C.bin file, will be consumed by the python script for the final validation. Your compiled binary (named “main”) should integrate properly with the provided python script. C/C++ code for reading and writing these binary files is given here.

Submission Rules 2: (If your files are not extracted correctly, you will earn 0):

Create a directory named “<kerberos_ID>_A0” and store the following files directly inside that directory: (Here, <kerbero_ID> is like cs1201111. Use your own ID.)

1. report.pdf containing the report document
2. data.csv containing the experimental data
3. main.cpp along with other code files (.c/.h/.cpp/.hpp) required for compilation
4. makefile required to compile your code. A nice introduction to create a basic makefile can be found here: https://www.geeksforgeeks.org/makefile-in-c-and-its-applications/. A binary executable file named “main” should get created after executing the “make” command in Linux.

Then ,execute the following linux command in the parent of this directory to create the zip file for submission:

```
zip -r <kerberos_ID>_A0.zip ./<kerberos_ID>_A0/
```

Finally, upload the resulting “<kerberos_id>_A0.zip” on Moodle.

Submission Rules 3:

As listed above, in addition to your code and makefile, your directory should also contain the raw data file and the report as described next.

Report:

Your report named report.pdf should consist of the following six sections:

1. Contains a plotted graph of execution time (Y-axis) vs matrix size (X-axis) keeping all six loop permutation as legends for experiment 1, where time is as reported by the Python script. (You should take the average of three runs for each data point.)

2. Same experiment as section 1, but total user times are now as reported by perf. Report if and why the times are different from section 2.
3. Same experiment as in Section 2, but the time is only the matrix multiplication time (not the file read and write times or memory allocation times).
4. Same experiment as in section 2, but now keeping cache-hit rate on the Y-axis.
5. Report the loop permutation which gave the best result in terms of best cache-hit rate (averaged across matrix sizes) and minimum execution time. Does the best permutation depend on the matrix size? Discuss why.
6. Using both the perf and gprof tools, report the percentage of time spent in each requested function (multiply, read, write, and main).

Raw Data

data.csv uses the csv format and contains the data of one experiment per line (averaged over 3 runs). An experiment is the requested matrix permutation and the matrix size. The columns in each line are: Permutation number, the row dimension of the input matrices (assuming all are square), total user time, user time in matrix multiplication, the percent of user time taken by matrix multiplication, total number of cache misses, cache-hit rate, instructions per cycle (all data from perf).

Grading:

You will be graded on the following:

1. The result matrix *C* that your code dumps in the <output_path>/mtx_C.bin file.
2. The findings that you have presented in your report and csv file.