**Computer Architecture - Spring 2018 - Lab #5**

**Cache Optimization**

Weighting: 5 Days

Due: Thursday 31, 2018 11:55pm

**You may work in pairs on this assignment.**

Access the assignment at:

[https://classroom.github.com/g/ShMMYpO4](https://www.google.com/url?q=https://classroom.github.com/g/ShMMYpO4&sa=D&ust=1526258774960000)

or on Polylearn

**Purpose**

To optimize data cache accesses for a well-studied, memory-bound application, and compare the impact on performance.

**Method**

Modify data structures and access patterns to exploit cache locality. Measure your optimizations using performance measurement tools (perf) for the ARM architecture.

**What to Do**

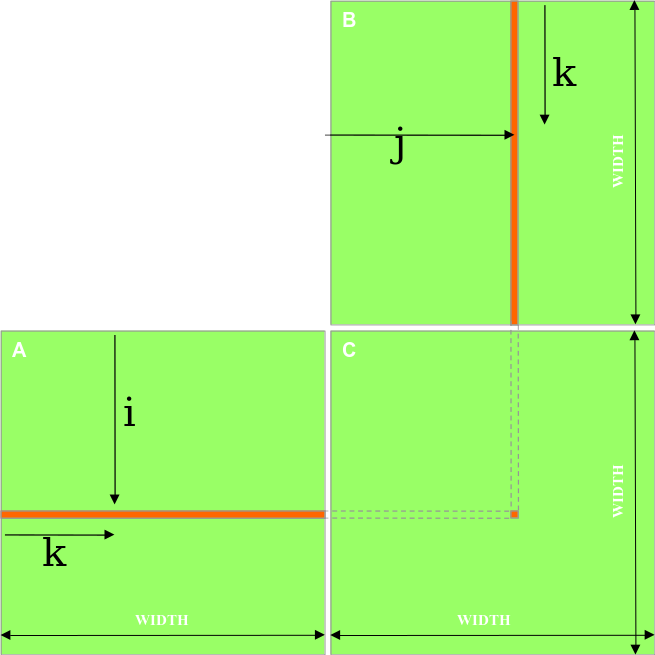
You will optimize the program, matmul. The first optimization will be to modify the memory layout of the second matrix such that it is in column-major order. The second optimization will be to implement a tiling algorithm to optimize cache locality. More details are below.

**matmul**

For the matmul program, you will modify two functions in matmul.cpp; matmul and Allocate2ndMatrix. The goal is to access the matrices in a way that exploits the spatial locality of the cache. You’ll do this both in the allocation/initialization, and in the actual computation.

The program computes the matrix multiplication of two 1024x1024 matrices, and writes the result to stdout. Run the program like ./mm > myout. A correct output file is provided so that you can compare your answer. Some small differences in floating point values (±0.01) might be observed, and are not a problem. A comparison script is provided that allows small tolerances.

The starter code you are given computes the multiplication: C = A x B, where A, B, and C are square matrices with n x n elements each, in the standard way: each element in C is the dot-product of a row in A and a column in B. This is depicted in the figure below. Unlike Lab 4, the matrices in this assignments are allocated *linearly* in a row-major order. This means that rows are adjacent in memory (the last element in row 0 is adjacent to the first element of row 1, etc.).



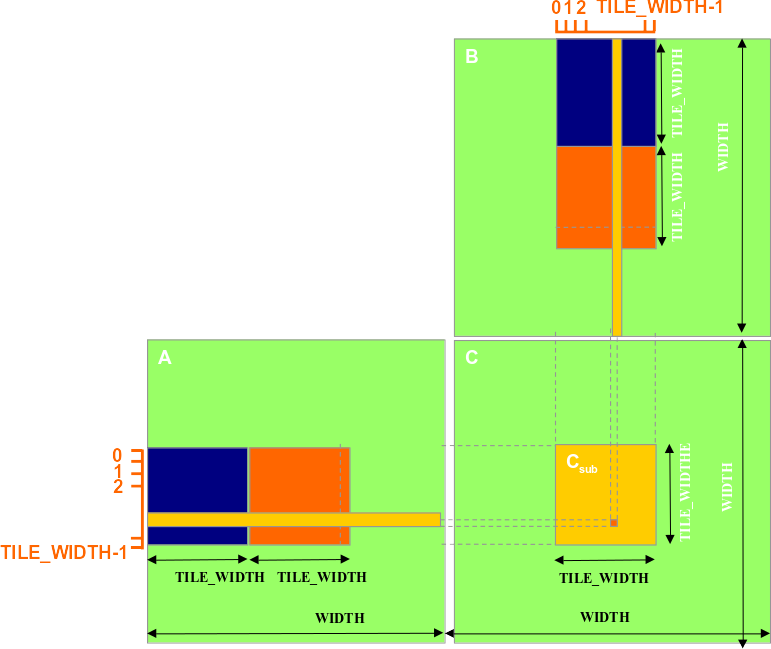
This is an O(n3) algorithm, where n is the width of each matrix.

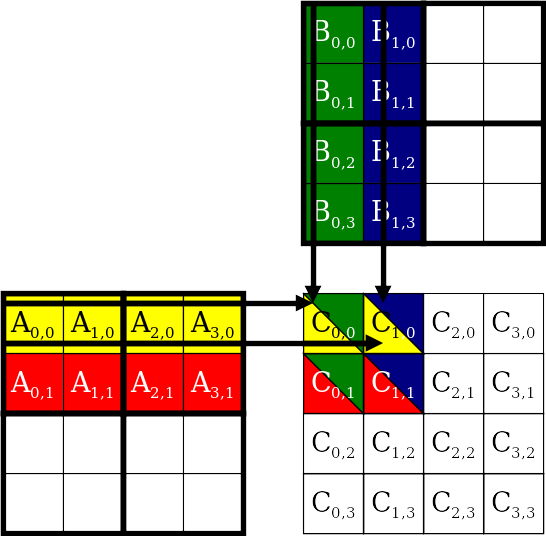
**Column-major order**

Your first optimization task is to convert matrix B from the default row-major order to column-major order, such that the elements in each column of B are sequential in memory. To do this, you’ll need to modify the matmul and Allocate2ndMatrix functions. **The computation must produce the same result** as the original code, but you  should see a significant reduction in execution time and cache accesses.

**Tiling (Full credit only given if run time decreases! Partial credit otherwise.)**

Your second optimization task is a bit more challenging. Starting with the column-major optimization above, you will next implement a *tiling* algorithm in the matmul function. Tiling, sometimes called blocking, is another technique to exploit memory locality. The idea is that you do all of the computation for one tile of the result matrix (C), before moving on to do any computation for the next tile. If the tile is constructed in such a way that a complete tile from A, B, and C can all fit in cache, then the number of cache misses will be significantly reduced compared to the non-tiled implementation. **For this lab, you will compare a tile size of 16x16 to a tile size of 32x32**. As before, the computation must produce the same result as the original code.





**Measurements**

You can measure the number of cache misses and cache-references using:

perf stat -e cache-misses -e cache-references ./mm > myout

To receive full credit on this lab, the number of cache accesses must be reduced for both optimizations.

**Testing**

A script is provided that compares all numbers in two files within a tolerance of 0.05. The test script (written in Python) is named compare.py, and is used as follows:

./compare.py <file1> <file2>

A file with correct output is provided named output.gold, that you can compare your output to.

**Record your run times, cache misses, and cache references.**

You should create two tables of the execution times, cache references, and cache misses. Complete the following tables:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Original Code** | **-O0** | **-O1** | **-O2** | **-O3** |
| **Runtime** | 203.502547986 | 158.443588311 | 157.050344757 | 157.693930635 |
| **Cache Misses** | 1,084,958,771 (4.790%) | 1,082,466,313 (33.172%) | 1,078,820,645 (33.080%) | 1,079,664,452 (33.115%) |
| **Cache References** | 22,648,387,050 | 3,263,165,751 | 3,261,202,103 | 3,260,346,442 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **With best compiler optimization level** | **Original** | **Column-major** | **Tiled 16x16** | **Tiled 32x32** |
| **Runtime** | 157.050344757 | 12.874084735 |  |  |
| **Cache Misses** | 1,078,820,645 (33.080%) | 3,837,421 (0.117%) |  |  |
| **Cache References** | 3,261,202,103 | 3,267,385,346 |  |  |

Explain your results in your report (e.g. correlate cache misses and cache references to measured performance). Compare your manual cache optimization to what the compiler is able to do.

**Web Resources**

[http://gcc.gnu.org/](https://www.google.com/url?q=http://gcc.gnu.org/&sa=D&ust=1526258774978000) - GCC Homepage

[https://perf.wiki.kernel.org/index.php/Tutorial](https://www.google.com/url?q=https://perf.wiki.kernel.org/index.php/Tutorial&sa=D&ust=1526258774978000) - perf tutorial

**What to Hand In**

An electronic copy of your report in PDF format. Please do not submit Microsoft Office documents. One report per group. Also submit a documented version of your code for both programs.

**Submission Instructions**

Submit the following files via GitHub:

        Your optimized source code,

        README.md with student names and GitHub account info included

Report in PDF format (no Docs!):

any scripts you wrote/used.