Canvas Group Name: Group_57
Student 1: Lawrence Lawson		Student 2: Vien Van

CMPE110 HA5: Locality Detective

Due Date: Thursday 05/24/18

In this assignment you will optimize a matrix multiplication kernel C = AB where A,B and C are NxN matrices, by optimizing for spatial and temporal locality. This assignment requires you to write C-code that needs to be submitted as part of your submission. Your submission needs to consist of this filled out pdf, the C-code file(s) as well as a README.txt file that explains how to compile and run the code. You will only receive code when we can reproduce your results with the submitted code. Code that does not compile will receive 0 points. Please use readable code and comments whenever useful. If we have problems understanding or running your code we might not be able to give you full credit so make it easy for us.

Use **N = 1024** for all assignments. You can find a skeleton of the matrix multiplication code here, which you need to use as a basis for this assignment:

https://drive.google.com/file/d/1I6xBbRO0aHcNq4LSiZcU2pcITdR8lY4C/view?usp=sharing

ALL RESULTS ARE REPORTED FROM PERFORMANCES ON SCHROEDINGER'S MACHINE

1) Unoptimized Kernel

Compile the kernel with gcc using optimization level -O3 and measure the execution time:

a) Execution Time: 4.677098 sec (1 Points)

b) What is the combined size of the 3 matrices in Bytes: $2^{23} * 3$ (1 Points)

c) How many bytes are read from memory for the matrices when executing the unoptimized kernel?

$2 * 1024^3 + 1024^2$ (1 Points)

2) Transpose Optimization

Add a method “transpose” that computes the transpose of a given matrix. Apply the transpose function to improve the execution time of the unoptimized kernel. To measure execution time it is not required to take into account the time of executing transpose (You can do it as part of the initialization). What execution time and speed up can you achieve? Verify that the code using the transposed matrix computes the same result matrix C as the base approach of 1.)

a) Execution Time: 0.580290 sec Speedup: $\approx 8x$ (4 Points)

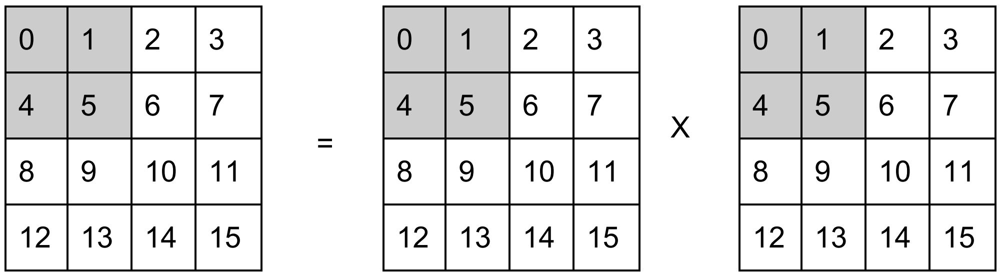
b) Given the achieved speedup, can you make any assumptions about the cache architecture of your system? Explain:

   In the unoptimized version of matrix multiplication, the function is loading one row from Matrix A, then N times for N rows in matrix B to get one column, to calculate one cell in the resulting matrix. But when using a transposed optimized multiplication, the cache loads an entire row from A and an entire row from B, then do calculations on the elements that are stored in the cache, leveraging Spatial locality, to access consecutive data that has already been cached, thus, avoiding cache misses within a single load. This results in the speedup that we see. We can conclude that the cache architecture leverages **spatial locality** , caching an entire row of a matrix load. We then take advantage of this spatial locality to compute dot product row-wise, making the execution time much faster.

(1 Points)

3. Blocking/Tiling Optimization

The given matrix multiplication kernel exhibits temporal locality that can be exploited via tiling. Tiling partitions the matrices into smaller sub-matrices and applies the matrix multiplication operation to these tiles instead of multiplying entire rows and columns at once.



a) Explain why the tiling optimization can improve performance. Describe the available temporal locality and how it can be exploited via tiling (2 Points)

By taking a small blocking size, such that the block can be held in registers, we can minimize the number of loads and stores in the program by accessing the most recently stored memory in the registers, leveraging temporal locality, which again improves performance.

b) Implement the tiling optimization for the matrix multiplication kernel. Make the number of tiles a configurable parameter (as a power of 2). Run the algorithm with all possible tile sizes (1, 2, 4, 8, 16, .. , 512, 1024 tiles per row/column of the original matrix) and measure the execution times. Verify that the code using the tiling approach computes the same result matrix C as the base approach of 1.) (8 Points)
This tiling algorithm is using transposed
TRANSPOSED

Blocking with transposed

1	17.594030
2	5.752116
4	2.388085
8	1.539556
16	0.97520
32	0.741958

64	0.706000
128	0.660861
256	0.635175
512	**0.608370**
1024	0.627312

Blocking without transposed

1	7.889752
2	2.658521
4	1.076763
8	0.692461
16	**0.662917**
32	1.065067
64	1.448130
128	1.495274
256	1.478395
512	1.473316
1024	3.968819

c) Given the performance results of 3.b) can you make assumptions about the cache sizes of your system? (2 Points)

We ran two different blocking algorithms, one blocking algorithm using transposed blocks, and the other using regular matrix multiplication.

The results we are seeing is that with transposed blocks, the optimal block size is 512, and for regular matrix multiplication, the optimal block size is significantly smaller at 16 blocks.

Side note, smaller array sizes may have additional overhead in the blocked version causes it to run slower than the non-blocked versions.

We can assume that performance is affected by the hardware we're running the algorithm on. We use tiling when we want to partition our matrix multiplication work, however some processors have different cache sizes, so the optimal block depends on the system's cache size. Furthermore, based on the results of the transposed blocking multiplication vs regular blocking multiplication, the cache will try to leverage the optimal **spatial and temporal locality.**

In other words, with a transposed multiplication scheme, the optimal block size will be bigger, because it is **leveraging spatial locality**. But for regular matrix multiplication without spatial locality, a smaller block size leverages **temporal locality** better.