Canvas Group Name: group\_58 Student 1: Matthew Wark, Mtwark@ucsc.edu Student 2: Kevin Le, kevle@ucsc.edu

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/1l6xBbRO0aHcNq4LSiZcU2pcITdR8lY4C/view?usp=sharing>

# 1) Unoptimized Kernel

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

1. Execution Time: **3.97 seconds** (1 Points)
2. What is the combined size of the 3 matrices in Bytes: 8 bytes \* 1024^2 \* 3 = **25,165,824** bytes (1 Points)

Reference: N x N = 1024^2 elements (1024 per matrix) \* 3 matrices \* 8 bytes (64 bit)

1. How many bytes are read from memory for the matrices when executing the unoptimized kernel?

Total of 1024^3 iterations in the for loops.

Each iteration accesses one element from A, one element from B, and one element from C.

Each element is 8 bytes

This gives (8 + 8 + 8)\*(1024^3)

= **25,769,803,776** bytes (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.)

1. Using -O3: Execution Time: **0.91 seconds** Speedup: **3.35x** (4 Points)
2. Given the achieved speedup, can you make any assumptions about the cache architecture of your system? Explain:

The cache is utilizing locality. The data stored in the cache takes advantage of spatial locality but without further deep investigation, we will not be able to determine the specific size of each cache line.

The system is storing sequential spaces in memory for our elements. Every access we take, we will grab multiple adjacent elements in memory which will reduce access time. We believe utilizing a transpose of our Matrix B reduces our access times by storing adjacent elements from adjacent memory addresses in the system’s cache.

(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

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

Matrix multiplication between two matrices A and B requires multiplying all of the elements of a single row, with a single element in every column of matrix B, and then adding the results together. This means that every element in every column is going to be needed n times, where n is the size of the height of matrix A. Since all of the elements from matrix B are being reused **n** times in the process of multiplying the matrices, we can exploit temporal locality.

For example, let’s say your cache isn’t large enough to hold every element in an arbitrary matrix B. Tiling helps resolve this issue by allowing you to break down matrix B into smaller sub-matrices, also known as tiles, which will fit in the cache. Classic matrix multiplication is then applied to each tile of the entire matrix, as if each tile is a regular element. This process allows for us to exploit temporal locality by reusing every element within each tile multiple times, causing cache hits.

1. 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)

Laptop And Server runs (included in this HW submission as 2 separate pdf files labelled accordingly):

<https://drive.google.com/open?id=1TBAwHafh-6qSIpahCnw4tP8kQMoncRDR>

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of Tiles** | **Original Runtime** | **Tiled Runtime** | **Speedup** |
| 1 | 9.70 seconds | 1.81 seconds | 436% |
| 2 | 9.89 seconds | 1.89 seconds | 422% |
| 4 | 9.71 seconds | 1.75 seconds | 455% |
| 8 | 9.60 seconds | 1.72 seconds | 456% |
| 16 | 9.77 seconds | 1.63 seconds | 497% |
| 32 | 9.54 seconds | 1.51 seconds | 533% |
| 64 | 9.72 seconds | 1.29 seconds | 648% |
| 128 | 9.76 seconds | 1.20 seconds | 707% |
| 256 | 9.72 seconds | 1.16 seconds | 736% |
| 512 | 9.66 seconds | 1.67 seconds | 476% |
| 1024 | 9.76 seconds | 2.95 seconds | 230% |

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

* The cache is only going to be large enough to hold a certain number of tiles (the fewer the tiles, the more full the cache will be since there are going to be more elements in each tile) and at some point while decreasing the number of tiles there should be too many elements per tile for the cache to hold and there should be a drop in performance.

The performance increase grew larger as the number of tiles used grew larger - up until using 256 tiles. We can assume that the cache is as full as we can make it using the given input values when we have tiles that are about 128 bytes large. 1024 rows / 256 tiles per row = 4x4 sized tiles. 4x4x8(bytes per element) = 128 bytes. The cache is not being well utilized, in terms of temporal locality, when using 512, or 1024 tiles per row/column, as is reflected by the sharp performance drop seen when going from 256 tiles to 512, or from 512 to 1024. When using less tiles than 256, the cache gets overfilled and the tiling method loses some of its effectiveness since certain elements in the tiles will be replaced in the cache prematurely, causing a larger miss rate.