The Effect of Cache Memory on Matrix Operations

Christian Kauten
Software Engineering and Computer Science
Auburn University
Auburn, AL, USA
jck0022@auburn.edu

***Abstract*—Modern computers use clever tricks to increase performance without incurring massive financial costs. One such performance improvement is seen in hierarchical memory where a series of progressively slower memory devices work together to reduce the latency induced by memory operations. Although these systems work well when used effectively, naive code can break the system, resulting in bad performance. These issues become highly apparent when looping over large matrices. Depending on the architecture of the cache, the ordering of the matrix in memory, and the code itself, initializing and performing actions on matrices could take a large amount of time. This paper explores the effects of cache on matrix operations using C-code. The results show that programmers need pay close attention to the design of their code to ensure proper utilization of the underlying hardware through novel means.**

***Index Terms*—Cache, Memory, Matrix, Transpose**

I. Introduction

A. Hierarchical Memory

Hierarchical memory utilizes a sequence of memory devices and heuristics to improve performance of memory operations. This sequence exists to balance performance and cost, allowing machines to take advantage of more expensive, but faster, memory devices. The machine can then leverage these devices for recurring or frequently used data to reduce latency.

1) Locality: In order to best utilize this hierarchy of devices, engineers employ two similar, but distinct heuristics based on *locality*. *Temporal locality* states that data used recently is likely to be used again soon. Without temporal locality, cache hits would rarely occur as new data from memory would always be needed. *Spatial locality* is the principle that data near recently used data is likely to be needed soon. This principle drives engineers to copy surrounding data to cache when a cache miss occurs for a memory address.

II. Methodology

A. Matrices

Matrices in C are stored in memory in row-major order. As such, they flatten out in memory as a sequential vector of rows. The most efficient way of *iteratively* traversing this matrix is sequentially iterating over each cell in each row in the vector. This is know as *row-wise* traversal. By properly utilizing spatial locality, it performs better than *column-wise* traversal. With large matrices, column-wise traversal will generate more cache misses as the algorithm wont portray the same degree of spatial locality. This is because each cell in a different row is a full matrix length of data types away from that cell in memory.

B. Transpose

The transpose operator, M^T, flips each value in a 2D matrix M by its i and j values.

III. Conclusions