Recursive Parallel Methods for Fast 2D Matrix Initialization and Transposition

Christian Kauten
Software Engineering and Computer Science
Auburn University
Auburn, AL, USA
kautenja@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 (Cache) 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. Recursive Initialization

In order to speed up initialization, the base matrices are encoded as piecewise functions based on the recursive series in the lower triangular matrix in each. This allows the matrices to initialize out of order as the incremental operator is no longer necessary. This encoding allows for a recursive divide-and-conquer initialization function. This function cuts the matrix into smaller sub-matrices and initializes them one at a time. Using a base case of an appropriately small matrix, the algorithm becomes “cache naive”. This means that it effectively uses the cache line without hardware specific blocking. Such a solution is more general than a blocking solution designed for a specific chipset.

1) Parallel Initializtion: The recursive algorithm lends itself well to parallelism as there is no need to lock for data access. The tux machines feature 8 physical cores that allow 16 contiguous threads. Each core has its own L1 and L2 cache, but share an L3 cache. As such, parallelism allows the software to better utilize the entirety of the cache on the machine. Because the machines are shared, the OS will have to schedule the processes across multiple users. As such, the program may run better on sequential runs as more resources are allocated by the OS for the single user account.

C. Transpose

The transpose operator, M^T, flips each value $v_{i,j}$ in a 2D matrix M by its i and j values. Many novel solutions exist to speed up the transpose operation. The fastest solution has a time complexity of $O(1)$. Instead of directly computing the transpose, the index can simply be inverted: $i \leftarrow j, j \leftarrow i$. This allows access to the transposed matrix with no change in memory. By inverting the index, this solution also inverts the optimal way to traverse the matrix from row-wise to column-wise. This makes optimizing cache usage of matrix operations more difficult for the programmer as the convention will switch from row-wise to column-wise in certain states.

D. *Recursive Transpose*

A recursive transpose operator divides a matrix into four even submatrices and computes their transpose. The bottom-left and top-right quadrants are then swapped. An alternative recursive solution performs the standard tranpose, but in a divide-and-conquer fashion. This latter solution allows effective cache line usage without additional swap operations or complex indexing.

1) Parallel Transpose: Much like the recursive intialization, the tranpose operation is also easily paralellized. This again allows for *more* cache and CPUs operating on the memory.

E. *Results*

With all improvements in place, the algorithm achieves a mean score of $\approx 4s$ using row-major ordering to initialize and transpose. Fig. 1 displays the mean times as it changes across the 5 trials of 7 experiments. The times vary slightly between sequential executions of the program as the OS allocates resources for the users.

Fig. 1. Mean Times to Initialize and Transpose a 40000x40000 Matrix

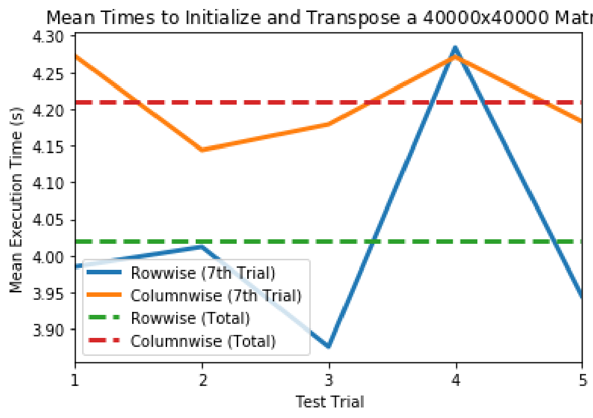


Fig. 2. Times to Initialize and Transpose a 40000x40000 Matrix

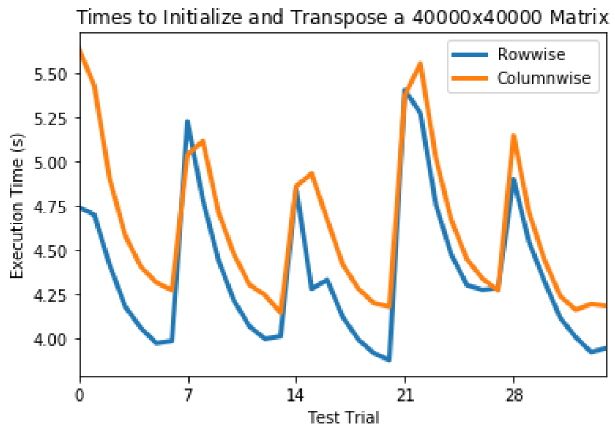
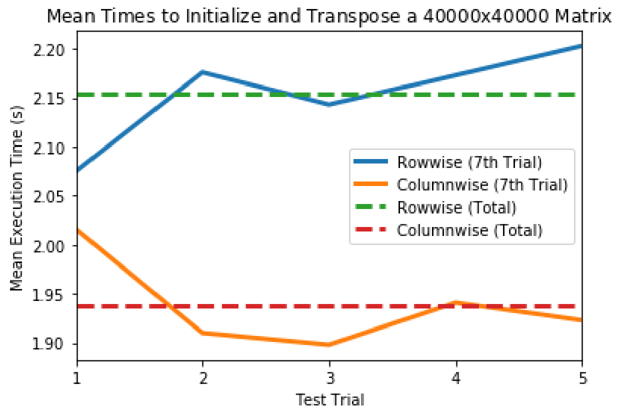


Fig. 2 shows the trend of mean across all tests in the 5 trials. Each trial demonstrates a gradual decrease in time with each additional test. This attributes to cache hits as the same memory is accessed by the same program during the same execution.

1) Compiler Optimizations: The GCC documentation reveals compiler flags for enabling compile-time optimizations. They offer a leveled system: `-O1`, `-O2`, and `-O3` that sequentially enable more features. A final extreme option `-Ofast` enables some addtional features relating to math and Fortran. The `-Ofast` flag produces the best improvement from the base results. Fig. 3 shows the means for the 5 trials of both row-wise and column-wise initialization and then transposition. The compile-time optimizations reduce both times by $\approx \frac{1}{2}$. Interestingly, column-wise operation outperforms row-wise across the board by $\approx 200ms$.

Fig. 3. Times to Initialize and Transpose a Matrix with Compiler Optimizations



III. Conclusions

This paper presents a recursive and parallel method to initialize and transpose a large square matrix in minimal time. A key feature of this recursive algorithm is a natural "cache naive" trait that allows it to exploit the cache line without explicit cache blocking. This coupled with a highly parallel architecture, like the 16-core Intel Xeon processors in the Tux machines, allows the algorithm to initialize and transpose a 40000x40000 matrix ($\approx$ 12.8GB) in 1.9 seconds ($\approx 6.7GB/s$). Although this time is fast, faster times certainly exist. Future research may investigate the integration of vector architectures like GPUs to speedup the initialization and transpose of matrices.