Timothy Holland

Matrix Experimentation Report

Introduction

The objective of this report is to compare the relative performance of a *naïve* transposition of a matrix with a *smart* divide-and-conquer cache-oblivious transposition algorithm. The naïve algorithm works by walking through the lower triangle of the matrix, swapping each item with corresponding item in upper triangle. Whereas the smart algorithm recursively breaks up the matrix until it reaches a single element, and only then does it swap an element. To do so, four identical experiments are performed across different simulated cache and block sizes: cache of 1024 items with 16 item blocks; cache of 8192 items with 64 item blocks; cache of 65536 items with 256 item blocks; cache of 65536 items with 4096 item blocks (these will be denoted in terms of concatenation of cache size prefixed by m and block size prefixed by b, for example the first test is denoted "m1024b16"). Each experiment compares each implementation in terms of a sequence of different matrix sizes mapped onto the average number of cache misses for transposing said matrix. I will consider the experiments two at a time, due to the importance of their comparison. Overall, I will conclude that the smart implementation allows for a low constant average cache misses compared to logarithmic function of the naïve implementation. However, I will also maintain that this difference in performance only holds for memory arranged with the tall cache assumption maintained.

m1024b16 and m8192b64 tests

To begin, I will consider the first two tests together as their results proof the same theorems, whilst introducing subtle differences to stimulate discussion. The first, and simplest test, compares the naïve and smart implementations on a cache of size 1024 items organised in 16 item blocks, mapping the average number of cache misses per item onto different matrix sizes:

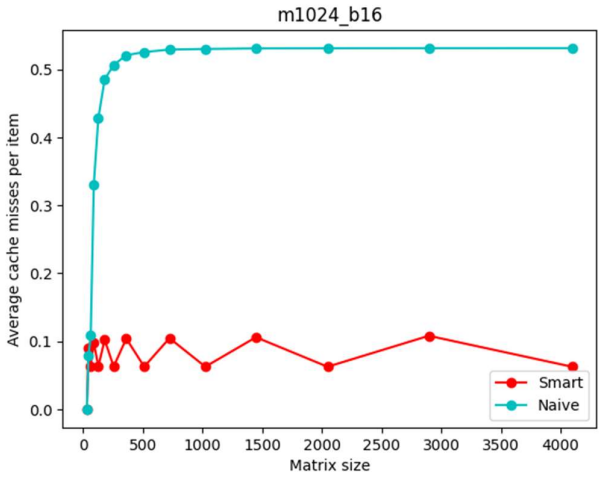

Figure 1: 1024 item cache, 16 item blocks

Whilst the second test (figure 2) compares the naïve and smart implementations on a cache size of 8192 items organised in 64 item blocks, mapping the average number of cache misses per item onto different matrix sizes:

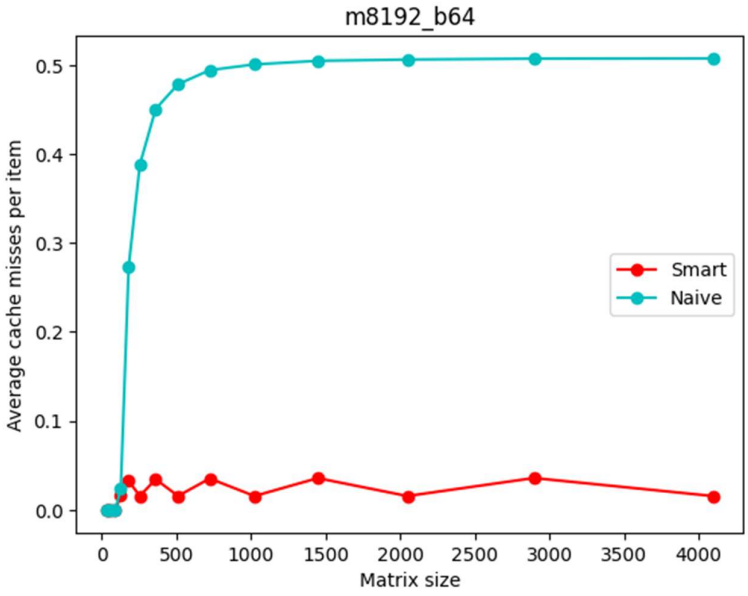

Figure 2: 8192 item cache, 64 item blocks

First, I will consider both tests generally, and then finish on some notable contrasting features. Most notably, across experiments, the two implementations differ in rate of change of the average cache misses as the matrix size grows, with the naïve implementation on a sharp logarithmic scale, whereas the smart implementation remains constant. To begin our analysis, and set up the rest of the experiments, it is important to consider why the massive discrepancy between these implementations.

The naïve implementation walks through iterates over lower triangle and swaps each item in with its equivalent in the upper triangle. The problem with this implementation is that because each triangle is looped over in a diagonal reflection, if we are accessing the lower triangle row-by-row, the upper will necessarily be accessed column by column. So, regardless of how blocks configure elements (row-by-row or column-by-column), this implementation results in a tight bound on the worst-case scenario, transfer $\Theta(N^2)$ blocks each time. Comparatively, the smart implementation appears constant, with a further constant overhead for matrix sizes which aren't a power of 2 (and thus aren't divisible by block size B).

The latter performance can be understood by considering the derivation of our cache-oblivious algorithm from a cache-aware model. As stated in the lectures, the optimal cache-aware algorithm splits the matrix into tiles of size d x d with smaller possibly rectangular tiles at border if N is not divisible by d. To transpose each tile, it must be swapped with its counterpart in the other triangle. To

understand the complexity on block transfers, consider d to be B. For the algorithm to be optimal, the cache must be able to hold two tiles at once, which avoids the above problem with the naïve implementation. Taking d to be B, each tile thus contains B^2 items, meaning that the memory must be satisfy the tall-cache property: $M \in \Omega(B^2)$. We thus arrive at the optimum block transfer complexity as stated in the lectures, $O(\frac{N^2}{B})$. Intuitively, this makes sense, as the greater the block size, the less transfers required. However, the tall-cache property must still be satisfied, otherwise for a greater number of tile swaps, a new block would have to be brought into the cache. This analysis translates into our cache-aware model, as we approximate the correct tile size by a series of recursive subdivisions (in this case to a singular element, which could be improved by approximating a greater sized matrix base case). As a result, the smart implementation remains constant across all matrix sizes for both experiments.

Despite this general analysis, three questions persist regarding our first two experiments. First, why do why lower size matrices (for example N = 32 or 45) result in similar performance across implementations? Simply, the cache is of size 1024 items, which is 32 x 32. Thus, all items can be fit into cache without the need for any block transfers. For the naïve implementation, this boost in performance degrades until the matrix is of size 256 in the first experiment, and 512 in the second. The worst case occurs here, for each required transposition a new block must be read into memory. Second, why does the second experiment's smart implementation have a constant factor performance increase over the first experiment? This fact can be reduced to the greater block size. As we are comparing performance on different matrix sizes, the greater the block size (whilst remaining consistent with the tall-cache property), the less block transfers and thus the lower average cache misses per the size of the matrix. Third, why is the worst case around 0.5 average cache misses per item? Put simply, every transpose operation is implemented on two items at a time, one in lower triangle, one in upper triangle. The worst case is when each transpose operation creates a cache miss (in triangle antithetical to the way the matrix is stored: bottom for column-by-column; top for row-by-row). Thus, discounting the diagonals which remain static, each pair costs one cache miss. The reason for the slight overhead is that the bottom triangle will also have cache misses when the end of a block has been reached.

m65536b256 and m65536b4096

Given our understanding of the fundamentals between each implementation from the first two experiments, I will now consider the remaining two experiments, with a focus on the essentiality of the tall-cache property. Our third experiment compares the naïve and smart implementations on a

cache of size 65536 items organised in 246 item blocks, mapping the average number of cache misses per item onto different matrix sizes:

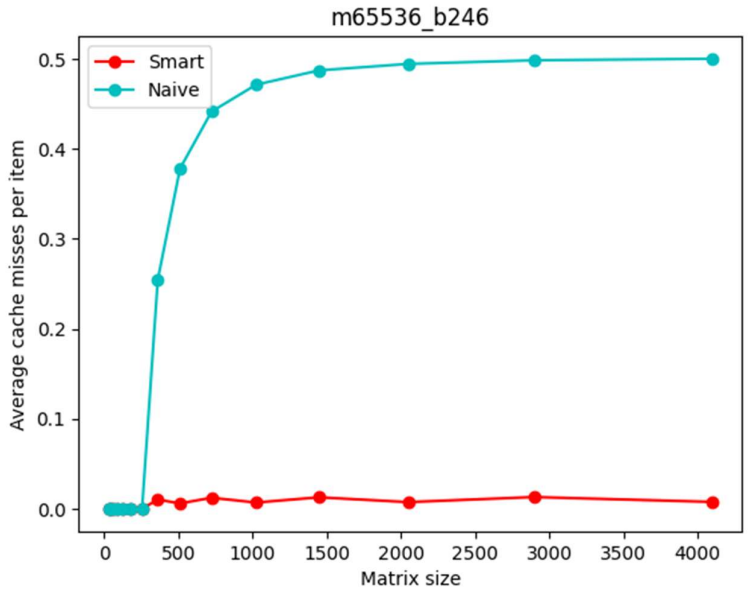

Figure 3: 65526 item cache, 264 item blocks

Whilst the fourth experiment, compares the naïve and smart implementations on a cache of size 65536 items organised in 4096 item blocks, mapping the average number of cache misses per item onto different matrix sizes:

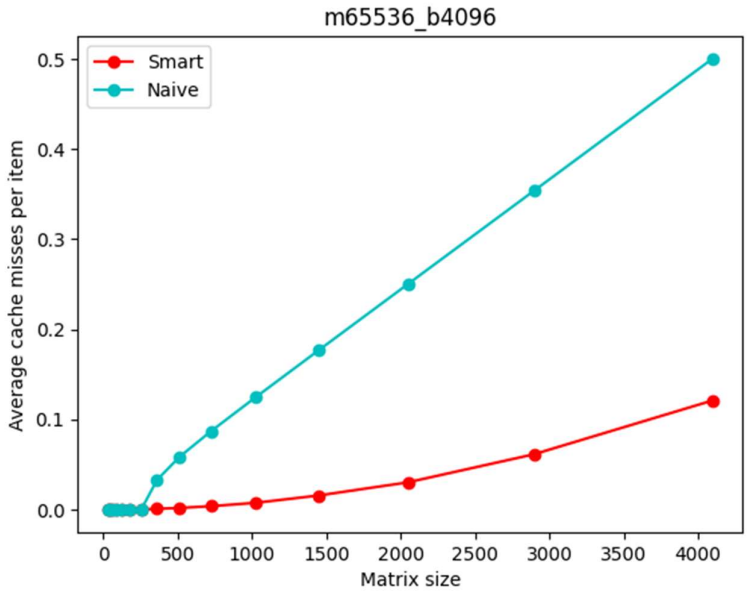

Figure 4: 65536 item cache, 4096 item blocks

Notably, the last two experiments differ only in block size, yet their results are strikingly different. To begin, the naïve implementation benefits from greater block size for all matrix sizes, appearing linear in average number of cache misses. We can attribute this to an above observation, that the greater block size means less cache misses are likely to occur. Even so, for 4096 sized matrices reduces to worst case block transfer of just above 0.5 as in all other experiments.

Perhaps more apparent is the performance degradation of the smart implementation. Despite comparable results in the third experiment, the greater block size significantly harms the performance, losing the constant factor misses to an increasing function. We can attribute this discrepancy by reconsidering our tall-cache property which stated cache size $M \in \Omega(B^2)$. The assumption holds in the third experiment ($256^2 = 65536$), but fails in the fourth experiment. The question becomes why the lack of tall-cache property results in the loss of optimality? The essential reason is that two blocks cannot fit into memory at the same time, meaning the foundation of the divide and conquer algorithm can no longer operate optimally. Specifically, as the size of the matrix grows, the likelihood of needing to bring in another block for each transpose swap operation increases. In other words, at any given time, one whole block will be present, and only part of another (which decreases as N increases). Thus, the result of increasing N increases the likelihood of cache misses, which can be seen in the results of the fourth experiment.