Computer Systems
Fall 2018
Lab Assignment 2: Reducing the Cache Miss Rate for Matrix Transpose
Assigned: October 2, Due: October 10, 20:00.

Nik (11058269), Tim (11848782)

1 Introduction

The goal of this lab is to illustrate the challenges in writing cache-friendly code. Specifically, it focuses on improving the cache performance of a given application - matrix transpose - by improving the locality of its memory accesses.

The lab is setup as follows: a "naive form" of matrix transposition is given, and it requires modification to reduce the number of cache misses for three specific matrix sizes: 32 x 32, 64 x 64, 61 x 67.

This report aims to describe the process of cache performance improvement, including the general approach, the challenges for each different version of the application, and a short discussion on the lessons learned.

2 Background

In this section we briefly introduce the necessary background and tools used for completing his lab.

Matrix transposition

Matrix transposition is an operation where the elements of a matrix are re-written in a different order, and has its main uses in linear algebra, and further in data analysis or computer graphics. The transformation is defined as follows: given a matrix A, of dimensions $M \times N$, and the transposed matrix A^T, of dimensions $N \times M$, $A^T_{i,j} = A_{j,i}, \forall i \in [1, N], j = [1, M]$.

The Memory Hierarchy and Caching

According to Bryant and O'Hallaron: "In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times. CPU registers hold the most frequently used data. Small, fast

cache memories nearby the CPU act as staging areas for a subset of the data and instructions stored in the relatively slow main memory."

Q1 (0.5p): Please state the principle of locality, and briefly explain the two different types of locality. Please note: in case you quote from a different source, you MUST use quotes and reference(s) accordingly. Caches improve performance because of the principle of locality, which states that: Programs tend to use data and instructions with addresses near or equal to those they have used recently. Locality is commonly divided into *Temporal Locality* and *Spatial Locality*. *Temporal Locality* refers to the same data being accessed or used in a small period of time, where *Spatial Locality* describes the use of data in close storage locations.

In this lab, we focus on the caches that serve the main memory, i.e., caches that work as an intermediate buffer between the CPU and main memory. Any memory access is, from the perspective of the cache, a **hit** - when the required data in already in the cache, or a **miss** - when the data is not in the cache. Furthermore, misses are further categorizes as: *cold*, *conflict*, and *capacity*, depending on why the data is missing from the cache.

The performance of a cache is judged in terms of its hit ratio - i.e., the number of accesses to the memory that end up as cache hits. A program with good locality will have a high hit ratio, which will translates in a high number of accesses to the cache, which are fast, and a low number of accesses to the main memory, which are slow.

There are three commonly-known cache organizations: Q2 (0.5p): Please briefly introduce/define the three well-known cache organizations, and state pro's and con's for each.

1. **Direct mapped cache**:
   In a direct mapped cache structure, the cache is organized into multiple sets with a single cache line per set. Based on the address of the memory block, it can only occupy a single cache line.
   Advantage: Placement is power efficient.
   Disadvantage: Lower cache hit rate. If the cache line is previously occupied, then the new data replaces the memory block in the cache.

2. **Set associative cache**:
   Set associative cache is a trade-off between Direct mapped cache and Fully associative cache.
   Advantage: Flexibility for replacement policies. Trade-off between direct mapped and fully associative.
   Disadvantage: Does not effectively use all the available cache lines in the cache.

3. **Full associative cache**:
   In a Fully associative cache, the cache is organized into a single cache set with multiple cache lines. A memory block can occupy any of the cache lines. Advantage: Provides flexibility when placing memory block. Better cache hit rate.
   Disadvantage: Placement is slow.

In this lab, we will focus on the L1 cache - i.e., the closest to the CPU - which is a direct mapped cache (i.e., E=1), with 32 sets (i.e., s = 5) and 32 bytes lines (i.e., b=5).

Valgrind

"Valgrind is an instrumentation framework for building dynamic analysis tools"[1], i.e., it can automatically instrument applications' code and allow for different data to be collected and analyzed to help understand a program's behavior and/or performance. For this lab, we use the cache profiler of Valgrind, which simulates the working of a given cache for a given application, and produces a detailed *trace of cache accesses*. By analysing this trace, one can see what has happened with the cache after every memory access. Such a fine-grained analysis may enable the performance engineer to improve the locality of the application, thus improving the hit ratio of the cache, which further improves the overall application performance.

Blocking/Tiling

Q3 (0.5p): Please explain what is tiling and how it can be achieved. Again, in case you quote other sources, bibliography references and, if necessary, quotes, must be used. Tiling divides a loop's iteration space into smaller blocks. This in turn results in data being held more efficiently in the cache until it is reused, thus reducing cache misses. In this lab, we will improve the performance of matrix transposition by using tiling as a main technique.

3 Approach and challenges

In this section we describe the general strategy we have devised to approach cache-performance optimization, and we further dive into the details of each of the three different matrix transposition exercises (i.e., sizes 32×32, 64×64, and 61×67).

General approach

The core of the matrix transposition application is the following loop:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
            AT[i,j] = A[j,i];
```

The main idea behind the cache-performance improvement in this lab is improving the locality of the application. To do so, we will reorder the memory accesses of the loops presented above. We will use two different techniques: tiling (as explained in Section 2) and buffering, which means using registers as temporary buffers to delay memory operations. .

The generic strategy to implement tiling is: Q4 (0.5p): Please explain, potentially using pseudocode, what was needed in the code to implement blocking? To separate the matrix into blocks 2 extra for-loops are added, the outer 2 for-loops go through the blocks by going to the first element of a block for each iteration, then the inner 2 for-loops go through the elements in that block, rows first.

[1] Valgrind: `http://valgrind.org/`

After tiling, the performance of the new application depends on ... Q5 (1.0p): Please comment on the impact of tiling on performance: are all tiling configurations (i.e., sizes) the same, performance wise? Is there a space-time tradeoff here?

Tiling causes less misses in the cache which improves locality which in turn improves performance. Not all tiling configurations are the same performance wise, for some problems bigger or smaller blocks are more optimal. Tiling improves speed but requires more space as more variables are used in the loops.

Q6 (1.0p): How can you compute the best tiling configuration for your application? Can you derive a formula to be used as an analytical model (i.e., an equation which, when you fill in the specific parameters of the problem at hand, can automatically compute the tile size) ?

For matrices with dimensions divisible by the cache size you can relatively easily determine the optimal size of the tiles, which is the cache size itself. For matrices for which this is not the case it is difficult to determine an optimal tile size by reasoning or calculations, in this case trial and error is usually the way to find the optimal tile size.

Solving the 32×32 transpose

Q7 (1.0p): Please briefly explain (with pseudo-code, too) the implementation for the 32×32 configuration. Clearly identify the tiling and the buffering.

Loop through blocks (size 8) of matrix A, rows first (tiling)
For each block loop through the elements of the block, rows first
if element is on diagonal of matrix A, store it temporary and store row or column (because row == column here)
if not diagonal put element in matrix B but rows and columns swapped (transpose of A)
at the end of an iteration through a row of a block, if current block is on the diagonal of matrix A, put temporary stored element in matrix B on stored place.

Solving the 61×67 transpose

Q8 (1.5p): Please briefly explain (with pseudo-code, too) the implementation for the 61×67 configuration. Clearly identify the tiling and the buffering. Please explain the difference(s) with the previous case. What was different - conceptually? Why was the change needed?

Loop through blocks (size 8) of matrix A, rows first (tiling)
For each block loop through the elements of the block and don't loop further than the dimensions of matrix A, rows first,
Put element in matrix B but rows and columns swapped (transpose of A)
Difference: The difference between this and 32x32 is that you need an additional check when looping through the rows in the blocks to make sure you do not loop further than the dimensions of the matrix, this is necessary because the dimensions are not divisible by the size of the blocks. Skipping diagonals in this case only gave an improvement of around 7 less misses which we determined not worth it.

Solving the 64×64 transpose

Q9 (1.5p): Please briefly explain (with pseudo-code, too) the implementation for the 64×64 configuration. Clearly identify the tiling and the buffering. Please explain the difference(s) with the previous case. What was different - conceptually? Why was the change needed?

Loop through blocks (size 4) of matrix A, rows first (tiling)
For each block loop through the elements of the block, rows first
if element is on diagonal of matrix A, store it temporary and store row or column (because row == column here)
if not diagonal put element in matrix B but rows and columns swapped (transpose of A)
at the end of an iteration through a row of a block, if current block is on the diagonal of matrix A, put temporary stored element in matrix B on stored place.
Difference: Block size is 4 instead of 8 to prevent overwriting in the cache when putting element in matrix B because with block size 8 the cache is not large enough so the first part of the data of the block in the cache gets overwritten when maximum capacity is reached.

4 Summary and Future Work

In this lab, we have improve the cache performance of matrix transposition by reordering its the memory accesses. To do so, we used a technique called tiling, and buffering (using registers).

Lessons Learned

Q10 (1.0p): What have you learned during this lab? Focus on your own knowledge and experience, and new skills (i..e, avoid "standard" statements). Consider both lab components: valgrind as a tool and tiling as a technique.

Mainly we have learned how to take the properties of the cache into consideration when writing a program to optimize the use of the cache and so also optimize the program. Tiling is a good way to improve the use of the cache by using all the elements in the cache before getting a new set of elements from the memory. Valgrind is a tool you can use to help optimize your program by giving information about what happens in the cache.

Future work

While we have improved the cache hit ratio, this lab misses a validation of the correlation between cache performance and the actual performance of the application. To further determine this correlation, ... Q11(1.0p): How would you check whether the performance of the application had improved?

The performance of the application can be checked by running the cache optimized application and the original application while measuring how long the execution takes. If the cache optimized application takes less time to execute then the performance has increased.

Moreover, we have focused on three specific sizes of the matrices. However, these techniques can/cannot be generalized ... Q12 (1.0p): Please discuss the potential generalization of the techniques used in this lab: which ones can be generalized, if any, and how? Referring to Q6, the tiling of some matrices, those with dimensions divisible by cache size, can be generalized by finding a formula that computes the tiling sizes.