Computer Systems 2022
Lab Assignment 3: Better caching for high performance
Assigned: Tuesday, October 4, Due: Thursday, October 13, 23:59.

Mick Cazemier, Gilian Honkoop

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, 61 x 67, and 64 x 64.

This report describes 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."

Caches improve performance because of the principle of locality, which states that: Q1 (1p): 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.

The principle of locality states that the addresses that programs access are often near or equal to recently used addresses. This concept can be split up into temporal- and spatial locality. With temporal locality, a recently accessed address is likely to be accessed again in the near future. With spatial locality addresses that lie close to each other are likely to be accessed around the same time frame.

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 is 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 ratio between the number of hits and the total number of memory accesses. A program with good locality will have a high hit ratio, which will translate 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 (1p): Please briefly introduce/define the three well-known cache organizations, and state one advantage and one disadvantage for each.

1. Direct mapped cache
   A direct mapped cache uses 1 line per set. This type of cache is very fast, however since every tag and set index is unique, the likelihood of conflict misses is increased.
2. E-way set associative cache
   E-way set associative cache uses E amount of lines per set. This reduces the amount of conflict misses, but it increases the amount of comparisons that need to be made since there are multiple blocks per set.
3. Fully associative cache
   With fully associative cache all lines are placed into the same set. Every cache line can be used since they are all in the same set, but every tag needs to be checked to see if something is already in the cache.

In this lab, we will focus on the performance of an L1 cache - i.e., the closest to the CPU - which is a direct mapped cache with 32 lines, each line having 32 bytes (in the notation of the Bryant and O'Hallaron book, we have E=1, s=5, and b=5). This cache is simulated in software.

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

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

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 (1p): Please explain what is tiling and why is it useful for improving cache performance. Again, in case you quote other sources, bibliography references and, if necessary, quotes, MUST be used.
Blocking is a technique where you divide a matrix into blocks, which are parts of the matrix. You then perform operations on these blocks instead of on the entire matrix. This is done to prevent having to access the main memory over and over, which takes a lot of time. It prevents this because the blocks are small enough that everything in a block fits in the cache.

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, how tiling is (generally) implemented.
Here you see an example of tiling, the optimal blocksize that is used here depends on the size of matrix A. It is essentially a transpose per block, until you've transposed the entire matrix.

```
for (i=0; i<N; i+=blocksize)
    for (j=0; j<N; j+=blocksize)
        for (p=i; (p<(i+blocksize)) && (p<N); p++)
```

```
for (q=j; (q<(j+blocksize)) && (q<N); q++)
        AT [p,q] = A[q,p];
```

After tiling, the performance of the new application depends on ... Q5 (0.5p): 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?

Different tiling configurations have different performances. Ultimately you want the smallest possible tiles and the lowest amount of total tiles, while the tile size still has to fit in the cache. Larger tiles or more tiles lead to more cache misses and thus more time needed to run the algorithm.

To compute the perfect tile size, we Q6 (1p): How can you compute the best tiling configuration for your application? Can you derive a closed-form expression/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) ?

In general it is ideal when an entire block can be loaded into the cache. The amount of rows that can be loaded in depend on the cache size and on how many elements are in each row. To get a good tile size you need to divide the cache size by the size of a row. The general formula thus becomes

$$tilesize = \frac{2^s \cdot E \cdot 2^b}{N \cdot sizeof(A[0][0])}$$

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, and explain how you chose the tile size.

```
blocksize = 8;
temp = 0;

for(i=0; i<M; i+=blocksize)
    for(j=0; j<N; j+=blocksize)
        for(p=i; (p<(i+blocksize)); p++)
            for(q=j; (q<(j+blocksize)); q++)
                if((p) == (q))
                    temp=A[p][q];
                else
                    B[q][p]=A[p][q];
            if(i==j)
                B[p][p]=temp;
```

To solve this transpose we divided the matrix up into blocks of 8×8. This block size was chosen so that an entire block could be loaded into the cache (256 / 32 = 8 rows). We used the tiling/blocking technique (as described in Q4) to transpose these blocks. We don't immediately transpose the main diagonal from A[0][0]

to A[N][M]. This is because when reading from the cache to fill matrix B, the positions on the diagonal point to the same address. This will cause conflict misses and throw the rest of the row from matrix A out of the cache. To prevent having to load the row again, we load the value of a diagonal position after the other spaces in that row have been filled. Since this block size allows us to load an entire row in the cache, we fill B row-wise, by having the inner loop looping over q.

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, and explain how you chose the tile size. Please explain the difference(s) with the previous case. What was different - conceptually? Why was the change needed?

```
for(i=0; i<M; i+=17)
    for(j=0; j<N; j+=17)
        for(p=j; (p<(i+17)) && (p<N); p++)
            for(q=i; (q<(j+17)) && q<M; q++)
                if((p)==(q))
                    temp=A[p][p];
                else
                    B[q][p]=A[p][q];
            if(i==j)
                B[p][p]=temp;
```

Solving this transpose was surprisingly easy, we applied tiling again and after some experimentation and reasoning we found the optimum tile size to be 17. This is because using a tile size of 17 allows you to have 4×4 tiles in total, which wouldn't be possible with a tile size of 16 or less. It also means that the incomplete tiles at the edges are as small as possible. The edge tiles would be larger with a tile size higher then 17, which would lead to more cache misses. This tile size is the perfect balance between a low amount of tiles and as small as possible tiles, this leads to the least amount of cache misses possible. However just tiling wasn't enough so we had to apply diagonal optimization just like we did in the 32×32 matrix, it works exactly the same as described above. Because M and N aren't exactly dividable by the tile size, we also had to add a check in the two most inner loops to check if the matrix size wasn't being bypassed. Before transposing the long side of A is represented by q and the short side by p. After transposing, these sides swap and thus we fill B by first filling the rows and then the columns. Thus the inner loop uses q to fill B row-wise. Doing this reduces the amount of rows that need to be loaded, and keeps the amount of columns that need to be loaded the same.

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, and explain how you chose the tile size. Please explain the difference(s) with the previous case. What was different - conceptually? Why was the change needed?

```
int blocksize = 8;
```

```
for(i=0; i<M; i+=blocksize)
    for(j=0; j<N; j+=blocksize)
        for(p=i; (p<(i+4)) && (p<M); p++)
            for(q=j+4; (q<(j+blocksize)) && q<N; q++)
                if((p)==(q-4))
                    temp=A[p+4][p];
                else
                    B[p][q]=A[q][p];
            if((i)==(j))
                B[p][p+4]=temp;

        for(p=i+4; (p<(i+blocksize)) && (p<M); p++)
            for(q=j+4; (q<(j+blocksize)) && q<N; q++)
                if((p)==(q))
                    temp=A[q][p];
                else
                    B[p][q]=A[q][p];
            if(i==j)
                B[p][p]=temp;

        for(p=i+4; (p<(i+blocksize)) && (p<M); p++)
            for(q=j; (q<(j+4)) && q<N; q++)
                if((p-4)==(q))
                    temp=A[p-4][p];
                else
                    B[p][q]=A[q][p];
            if((i)==(j))
                B[p][p-4]=temp;

        for(p=i; (p<(i+4)) && (p<M); p++)
            for(q=j; (q<(j+4)) && q<N; q++)
                if((p)==(q))
                    temp=A[p][p];
                else
                    B[p][q]=A[q][p];
            if(i==j)
                B[p][p]=temp;
```

Unfortunately solving the transpose of the 64×64 matrix took a lot more effort then solving the other two. First of all we again used tiling, but this time we used tiles of 8×8which we divided up further into tiles of 4×4. This is so that we can use small tiles, but also make use of the fact that 8 numbers fit in the cache. Then we perform our operations on these four by four matrices, but we do this in a specific order per 8×8 tile. If you picture the transformation like this:
X Y —X'Z'

Z W —Y'W'
Then we first transform Z into Z', which loads W into the cache. Then we transform W into W'. After this we transform Y into Y'. and lastly we transform X into X'. This makes sure we waste as little addresses that are loaded into the cache as possible. We immediately use the addresses that are loaded into the cache. We also take care of the diagonal like we did in the earlier problems.

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 (1p): 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.

During this lab we have extensively experimented with and thought about ways to optimize transposing matrices using tiling/blocking. We learnt how cache loads in rows and how we can make use of this by transposing and storing certain blocks before others. It is efficient to use as much data stored in the cache as possible, before overriding it by loading in new addresses. We also learnt why transposing the diagonal prevents additional conflict and cold misses. While we did make use of the Valgrind framework, we didn't really learn how it works. This is because there wasn't really a need for us to learn anything about it. The given commands to run the tests were enough for us.

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, we would ... Q11(0.5p): How would you check whether the performance of the application had improved?

A way to check the actual performance is to measure the time it takes to transpose different matrix. You could optimize cache performance, but if you have to do a lot of intensive calculations it can slow the overall run time.

Moreover, we have focused on three specific sizes of the matrices. However, these techniques can/cannot be generalized ... Q12 (0.5p): Please discuss the potential generalization of the techniques used in this lab: which ones can be generalized, if any, and how? How would you recommend the generalization be formulated as a future work direction?

Tiling/blocking is generalizable, since you can always divide a matrix into blocks, and those blocks into even smaller blocks until you reach your wanted blocksize. The diagonalization technique where we transpose the main diagonal after each row is also generalizable. If you take each block on its own, it also has a diagonal on which this technique can be applied. And every matrix and block has a diagonal.