CS-341L, Fall 2018
Cache Lab: Understanding Cache Memories

Assigned: Tuesday, Oct. 29, 2019
Due date for Part B: Nov. 26th 11:59 pm

Edited to show only updates for part B

1 Logistics

This is an individual project. You must run this lab on a 64-bit x86-64 machine. Please start early. Make sure you read the whole statement of the assignment and understand it fully. For clarifications you can attend the office hours of your instructor or teaching assistants or you may send email to any of them, but ask for these clarifications early so that there is enough time to solve any problems before the deadline and so that you have plenty of time to invest in the project.

2 Overview

This lab will help you understand the impact that cache memories can have on the performance of your C programs.

The lab consists of two parts. In the first part you will write a small C program (about 200-300 lines) that simulates the behavior of a cache memory. In the second part, you will optimize a small matrix transpose function, with the goal of minimizing the number of cache misses.

3 Downloading the assignment

Log in to any of the cs machines. In the `/nfs/faculty/soraya/CS341-Fall2019` directory in the cs machines you will find a file called `cachelab-handout.tar`, copy this file to a **protected (only you may read, write or execute) Linux directory** in which you plan to do your work. Then give the command:

```
    linux> tar xvf cachelab-handout.tar
```

This will create a directory called `cachelab-handout` that contains a number of files. You will be modifying two files: `csim.c` and `trans.c`. To compile these files, type:

```
linux> make clean
linux> make
```

By now, after doing part A you already downloaded the files needed. We will only update a couple of files for part B you will find them on Learn.

WARNING: Do not let the Windows WinZip program open up your `.tar` file (many Web browsers are set to do this automatically). Instead, save the file to your Linux directory and use the Linux `tar` program to extract the files. In general, for this class you should NEVER use any platform other than Linux to modify your files. Doing so can cause loss of data (and important work!).

4 Description

In Part A of this project (described in section 4.2), you will implement a cache simulator. In Part B you will write a matrix transpose function that is optimized for cache performance (see section 4.3). For this first assignment concentrate on Part A, we will publish an update for Part B to eliminate one of the tests.

Omitting the description necessary for part A.

4.1 Reference Trace Files

The content of this section was omitted.

4.2 Part A: Writing a Cache Simulator

This section was also omitted.

4.3 Part B: Optimizing Matrix Transpose

In Part B you will write a transpose function in `trans.c` that causes as few cache misses as possible.

Let A denote a matrix, and A_{ij} denote the component on the ith row and jth column. The *transpose* of A, denoted A^T, is a matrix such that $A_{ij} = A^T_{ji}$.

To help you get started, we have given you an example transpose function in `trans.c` that computes the transpose of $N \times M$ matrix A and stores the results in $M \times N$ matrix B:

```
char trans_desc[] = "Simple row-wise scan transpose";
void trans(int M, int N, int A[N][M], int B[M][N])
```

The example transpose function is correct, but it is inefficient because the access pattern results in relatively many cache misses.

Your job in Part B is to write a similar function, called `transpose_submit`, that minimizes the number of cache misses across different sized matrices:

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N]);
```

Do *not* change the description string ("`Transpose submission`") for your `transpose_submit` function. The autograder searches for this string to determine which transpose function to evaluate for credit.

You may use the provided simulator to solve the cache optimization problems in this part.

Part B is due on November 26th, see details in section 7.

Programming Rules for Part B

- Include your name and netID in the header comment for `trans.c`.
- Your code in `trans.c` must compile without warnings to receive credit.
- You are allowed to define at most 12 local variables of type `int` per transpose function.[1]
- You are not allowed to side-step the previous rule by using any variables of type `long` or by using any bit tricks to store more than one value to a single variable.
- Your transpose function may not use recursion.
- If you choose to use helper functions, you may not have more than 12 local variables on the stack at a time between your helper functions and your top level transpose function. For example, if your transpose declares 8 variables, and then you call a function which uses 4 variables, which calls another function which uses 2, you will have 14 variables on the stack, and you will be in violation of the rule.
- Your transpose function may not modify array A. You may, however, do whatever you want with the contents of array B.
- You are NOT allowed to define any arrays in your code or to use any variant of `malloc`.

5 Evaluation

This section describes how your work will be evaluated. The full score for this lab is 60 points:

- Part A: 27 Points
- Part B: 26 Points
- Style: 7 Points

[1]The reason for this restriction is that our testing code is not able to count references to the stack. We want you to limit your references to the stack and focus on the access patterns of the source and destination arrays.

5.1 Evaluation for Part A

Content omitted.

5.2 Evaluation for Part B

For Part B, we will evaluate the correctness and performance of your `transpose_submit` function on three different-sized output matrices:

- 32×32 ($M = 32$, $N = 32$)
- 64×64 ($M = 64$, $N = 64$)
- 61×67 ($M = 61$, $N = 67$)

In the Fall 2019 semester, you are required to transpose only the 32×32 and the 61×67 matrices. The transpose of the 64×64 matrix will be extra credit if you decide to do it.

5.2.1 Performance (26 pts)

For each matrix size, the performance of your `transpose_submit` function is evaluated by using `valgrind` to extract the address trace for your function, and then using the reference simulator to replay this trace on a cache with parameters ($s = 5$, $E = 1$, $b = 5$).

Your performance score for each matrix size scales linearly with the number of misses, m, up to some threshold:

- 32×32: 8 points if $m < 300$, 0 points if $m > 600$
- 64×64: 8 points if $m < 1,300$, 0 points if $m > 2,000$
- 61×67: 10 points if $m < 2,000$, 0 points if $m > 3,000$

Your code must be correct to receive any performance points for a particular size. Your code only needs to be correct for these three cases and you can optimize it specifically for these three cases. In particular, it is perfectly OK for your function to explicitly check for the input sizes and implement separate code optimized for each case.

NOTE: as stated in section 5.2 the 64x64 matrix is not required, therefore the maximum total value for the performance of the matrices is 18 points and not 26.

5.3 Evaluation for Style

There are 7 points for coding style. These will be assigned manually by the course staff. We will provide some Style guidelines in Learn but the minimum you need to comply with is what we asked for in the first programming assignment: "Your solutions should be as clean and straightforward as possible. Your comments should be informative, but they need not be extensive."

The course staff will inspect your code in Part B for illegal arrays and excessive local variables.

6 Working on the Lab

For the Fall 2019 semester we are leaving the section titles so that the numbers do not change, but we are focusing on part B only for this second part of the project.

6.1 Working on Part A

Content omitted.

6.2 Working on Part B

We have provided you with an autograding program, called `test-trans.c`, that tests the correctness and performance of each of the transpose functions that you have registered with the autograder.

You can register up to 100 versions of the transpose function in your `trans.c` file. Each transpose version has the following form:

```
/* Header comment */
char trans_simple_desc[] = "A simple transpose";
void trans_simple(int M, int N, int A[N][M], int B[M][N])
{
    /* your transpose code here */
}
```

Register a particular transpose function with the autograder by making a call of the form:

```
registerTransFunction(trans_simple, trans_simple_desc);
```

in the `registerFunctions` routine in `trans.c`. At runtime, the autograder will evaluate each registered transpose function and print the results. Of course, one of the registered functions must be the `transpose_submit` function that you are submitting for credit:

```
registerTransFunction(transpose_submit, transpose_submit_desc);
```

See the default `trans.c` function for an example of how this works.

The autograder takes the matrix size as input. It uses `valgrind` to generate a trace of each registered transpose function. It then evaluates each trace by running the reference simulator on a cache with parameters ($s = 5$, $E = 1$, $b = 5$).

For example, to test your registered transpose functions on a 32×32 matrix, rebuild `test-trans`, and then run it with the appropriate values for M and N:

```
linux> make
linux> ./test-trans -M 32 -N 32
Step 1: Evaluating registered transpose funcs for correctness:
func 0 (Transpose submission): correctness: 1
func 1 (Simple row-wise scan transpose): correctness: 1
func 2 (column-wise scan transpose): correctness: 1
func 3 (using a zig-zag access pattern): correctness: 1

Step 2: Generating memory traces for registered transpose funcs.

Step 3: Evaluating performance of registered transpose funcs (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1766, misses:287, evictions:255
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151
func 2 (column-wise scan transpose): hits:870, misses:1183, evictions:1151
func 3 (using a zig-zag access pattern): hits:1076, misses:977, evictions:945

Summary for official submission (func 0): correctness=1 misses=287
```

In this example, we have registered four different transpose functions in `trans.c`. The `test-trans` program tests each of the registered functions, displays the results for each, and extracts the results for the official submission.

Here are some hints and suggestions for working on Part B.

- The `test-trans` program saves the trace for function i in file `trace.f`i.[2] These trace files are invaluable debugging tools that can help you understand exactly where the hits and misses for each transpose function are coming from. To debug a particular function, simply run its trace through the reference simulator with the verbose option:

  ```
  linux> ./csim-ref -v -s 5 -E 1 -b 5 -t trace.f0
  S 68312c,1 miss
  L 683140,8 miss
  L 683124,4 hit
  L 683120,4 hit
  L 603124,4 miss eviction
  S 6431a0,4 miss
  ...
  ```

[2]Because `valgrind` introduces many stack accesses that have nothing to do with your code, we have filtered out all stack accesses from the trace. This is why we have banned local arrays and placed limits on the number of local variables.

- Since your transpose function is being evaluated on a direct-mapped cache, conflict misses are a potential problem. Think about the potential for conflict misses in your code, especially along the diagonal. Try to think of access patterns that will decrease the number of these conflict misses.
- Blocking is a useful technique for reducing cache misses. See the following url

  ```
  http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf
  ```

  for more information.

6.3 Putting it all Together

We have provided you with a *driver program*, called `./driver.py`, that performs a complete evaluation of your simulator and transpose code. This is the same program your instructor uses to evaluate your handins. The driver uses `test-csim` to evaluate your simulator, and it uses `test-trans` to evaluate your submitted transpose function on the three matrix sizes. Then it prints a summary of your results and the points you have earned.

To run the driver, type:

```
linux> ./driver.py
```

We are using the same driver program, as the one for part A, but if you are not doing the 64×64 matrix, you will don't get a score for that transpose.

7 Submitting Your Work

Each time you type `make` in the `cachelab-handout` directory, the Makefile creates a tarball, called `userid-handin.tar`, that contains your current `csim.c` and `trans.c` files. Part B, consists of the `trans.c`, but you will submit the .tar file for this final part.

By the second deadline of 11/26/2019, you will submit your final `userid-handin.tar` file in UNM Learn, in the place for that second portion specifically (it is a different assignment on Learn).

IMPORTANT: Do not create the handin tarball on a Windows or Mac machine, and do not handin files in any other archive format, such as `.zip`, `.gzip`, or `.tgz` files.