CS 213, Spring 2000
Lab Assignment L4
Assigned: March 23
Due: April 12, 11:59PM

Logistics

Ted Wong (`tmwong@cs.cmu.edu`) is the lead person for this assignment. The files for this assignment can be retrieved from:

`/afs/cs.cmu.edu/academic/class/15213-s00/L4/L4.tar`

Once you've copied this file into a (private) directory, run the command `tar xvf L4.tar`, and fill in your team information in the structure at the beginning of the file `mxm.c`. When you have completed the lab, you will hand in two files, `mxm.c` and `mxm_through.c`, that contain your solution. Your grade will be determined by how well your solutions perform compared to an optimized reference implementation.

Introduction

The purpose of this assignment is to gain experience in optimizing code for a memory intensive application. Consider a procedure to multiply two matrices A and B and store the result in matrix C. A naive implementation might use nested loops like this:

```
void
mxm(array_t A, array_t B, array_t C){
  int i, j, k;

  for (i = 0; i < ARRAY_ROWS(A); i++){
    for (j = 0; j < ARRAY_COLS(B); j++){
      ARRAY_ELEM(C, i, j) = 0;
      for (k = 0; k < ARRAY_COLS(A); k++){
        ARRAY_ELEM(C, i, j) += ARRAY_ELEM(A, i, k) * ARRAY_ELEM(B, k, j);
      }
    }
  }
}
```

where the arguments to the procedure are C `typedef structss` representing A, B, and C. Note that A and B may have arbitrary dimensions - in fact, they may not even be square matrices!

Dimensions of A	Dimensions of B	Hit Ratio	Throughput (MB/s)
7×7	7×7	1.000	6.03
32×32	32×32	1.000	1.59
64×64	64×64	0.997	0.78
103×103	103×103	0.996	0.48
128×128	128×128	0.972	0.29
256×256	256×256	0.972	0.11
61×253	253×253	0.996	0.19
137×57	57×193	0.994	0.83

Table 1: Measured cache-hit ratio and throughput of the naive matrix-multiply implementation

Table 1 summarizes the performance of the naive implementation in terms of both cache-hit ratio and throughput. We determine the latter by measuring the length of time t it takes to find C, and compute $((M \times N \times 4 \times 10^{-6})/t)$, where M and N are the dimensions of C. We see that the throughput can vary widely for different input matrix sizes, even if the cache-hit ratios are similar.

As you have seen in lecture, we can improve upon the naive implementation by changing the order of the loops, and by using blocking to exploit cache locality. However, to achieve the highest throughput, you will discover we also need to employ other optimization techniques such as loop unrolling.

Part I: Optimizing Simulated Cache Performance

The best way to improve the performance of the implementation is to first focus on its cache behavior. To measure the cache behavior more precisely, we will run the implementation against a simulated 16 KB 4-way associative cache with 32 byte lines. This is the same configuration as the L1 cache of the Pentium system you will target Part II to, so a good implementation here will serve you well when optimizing throughput.

We instrument the matrix-multiply implementation to capture memory accesses using `cacheprof`, which is a public-domain cache simulator (`http://www.cacheprof.org/`). `cacheprof` instruments assembly code to capture the source (destination) addresses of read (write) instructions, and uses them to count hits and misses in a simulated cache.

Your task is to modify the code for the procedure `mxm(...)` in `mxm.c` to improve its cache performance. We provide you with eight test matrix multiplications to measure your performance, which include both square and non-square matrices, and non-power-of-two sized matrices. Make sure that your solution is correct for arbitrary matrices - the test cases we have provided are a subset of the ones we will use to verify the correctness of your implementation!

To check your progress you can compile and run the code on your own machine with the command `make`, and run the driver program for a specific test case with the command `driver [test_num]`. The program will output cache performance statistics (and throughput results that are pertinent to Part II). You can also run all the test cases using `driver_all`.

If you are having problems debugging your code, you should change the `Makefile` by un-commenting the line:

```
CFLAGS=-Wall -g $(DEFINES)
```

commenting out the lines:

```
mxm.o: mxm.c
$(CACHEPROF) $(CACHEPROFFLAGS) $(CC) $(CFLAGS) $(INCDIR) -c -o $*.o $<
```

and using `make clean; make`. This will disable `cacheprof` and include debugging information needed by `gdb`.

When you are done with Part I, you may want to change the `Makefile` by commenting out the lines:

```
mxm.o: mxm.c
$(CACHEPROF) $(CACHEPROFFLAGS) $(CC) $(CFLAGS) $(INCDIR) -c -o $*.o $<
```

and using `make clean; make`, so that you do not have to wait for the simulation to complete when working on Part II.

Part II: Optimizing Measured Throughput

Your task here is to modify the code for the procedure `mxm_through(...)` in `mxm_through.c` so that it has the highest throughput you can achieve for arbitrary-sized matrices. Improving the performance here requires a fair amount of low-level code optimization. For example, you may want to unroll loops so that the inner loop performs the multiplication operations for a number of matrix elements, thus amortizing the loop overhead.

You may find it helpful to change the `Makefile` by un-commenting the lines:

```
mxm_through.o: mxm_through.c
$(CACHEPROF) $(CACHEPROFFLAGS) $(CC) $(CFLAGS) $(INCDIR) -c -o $*.o $<
```

and using `make clean; make`. `cacheprof` will then instrument `mxm_through(...)`, allowing you to examine its cache performance as you optimize it. Remember that the instrumentation will reduce your throughput, so when you are satisfied with the cache performance you should re-comment those lines.

You might notice your throughput results varying between different runs, since you are sharing machines with other groups working at the same time. To get better results, you can submit your implementation to the class server using the command `make submit NAME=yourname`. The measured results will be returned to you by electronic mail, and will also be shown on a class-wide WWW page summarizing the cache performance of `mxm(...)` and the throughput of `mxm_through(...)`.

Rules

You may write any code you want, as long as it satisfies the following:

- It must be in ANSI C. You may not use any embedded assembly language statements.
- It must be correct.
- It must not print any extraneous information.

- It must not interfere with the cache simulation or time measurement mechanism. In particular, we will penalize spurious copies designed to artificially boost your cache-hit ratio - we can check this by examining the number of total memory references your code makes.

We have provided helper macros is `defs.h` for accessing array elements. We recommend that you use these macros, since the arrays are actually implemented using C `typedef structs`, but if you chose to you can manipulate the array elements directly.

You are allowed to define other macros, global variables, and procedures in `mxm.c` and `mxm_through.c`. If you modify any other files, make sure your `mxm.c` and `mxm_through.c` still run with the original versions.

Evaluation

We are currently preparing an optimized reference implementation. When it is ready, we will compute the harmonic mean $\overline{R_{opt}}$ of the cache-hit ratios for the 8 test cases, and the harmonic mean $\overline{T_{opt}}$ of the throughputs. We will publish these numbers on the class bulletin board.

Your grade will be based on the following:

- Correctness: You will get NO CREDIT for buggy code! This includes code that correctly multiplies the provided test matrices, but incorrectly multiplies other matrices.
- Cache-hit ratio: You will get full credit for Part I if your implementation is correct and achieves a mean cache-hit ratio of at least $\overline{R_{opt}}$. You will get partial credit for a correct implementation that does better than the supplied naive one.
- Throughput: You will get full credit for Part II if your implementation is correct and achieves a mean throughput of at least $\overline{T_{opt}}$. You will get partial credit for a correct implementation that does better than the supplied naive one.

Hand In

Hand in your code using the command `make handin NAME=yourname`, where `yourname` is the Andrew ID of one of the team members. You can submit updated versions with the command `make handin NAME=yourname VERSION=XX`, where `XX` is the version number, i.e., 2, 3, ... Only your highest numbered version submitted before the deadline will be graded.