CS 213, Fall 2000
Lab Assignment L3
Assigned: October 12
Due: November 1, 11:59PM (Note date change)

Logistics

Bianca Schroeder (bianca@cs.cmu.edu) is the lead person for this assignment. The files for this assignment can be retrieved from

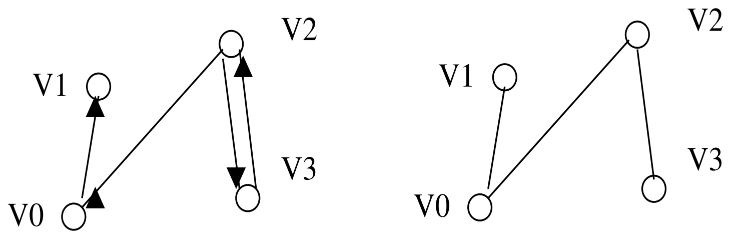
/afs/cs.cmu.edu/academic/class/15213-f00/L3/L3.tar

Once you've copied this file into a (private) directory, run the command `tar xvf L3.tar` and fill in your team information in the structure at the beginning of the file `conversion.c`. When you have completed the lab, you will hand in two files: `conversion.c` and `conversion_through.c` that contain your solution. For the first part you will be graded on the performance of your code for the routine `good_cache` in `conversion.c`. For the second part you will be graded on the performance of your code for the routine `good_throughput` in `conversion_through.c`. Your grade will be determined by how well your routines perform compared to an optimized reference solution.

Introduction

The purpose of this assignment is to gain experience in optimizing code for a memory intensive application. We consider the problem of converting a directed graph g into its undirected counterpart g'. g' has an edge from vertex u to vertex v iff there is an edge from u to v or from v to u in the original graph g. The graph g is represented by its *adjacency matrix* G as follows. If N is the number of vertices in g then G is an $N \times N$ matrix and its entries are all either 0 or 1. Suppose the vertices of g are named $v_0, v_1, v_2, ..., v_{N-1}$. Then $G[i][j]$ is 1 if there is an edge from v_i to v_j and 0 otherwise. Figure 1 shows an example for a directed graph g, the corresponding undirected graph g' and the adjacency matrices of the two graphs. Observe, that the elements on the diagonal of an adjacency matrix are always 1 and that the adjacency matrix of an undirected graph is symmetric.

The adjacency matrix of a directed graph can be converted into the adjacency matrix of the corresponding undirected graph in a simple nested loop:



$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

Figure 1: The graphs g (left) and g' (right) and their adjacency matrices

```
int i, j;
for (i = 0; i < dim; i++)
  for (j = 0; j < dim; j++)
    G[RIDX(j,i,dim)] = G[RIDX(j,i,dim)] || G[RIDX(i,j,dim)];
```

where the arguments to the procedure are a pointer to the adjacency matrix (`G`), as well as the matrix size N (`dim`). The macro `RIDX` is defined as follows:

```
#define RIDX(i,j,n) ((i)*(n)+(j))
```

This code for converting g into an undirected graph goes through the entries of `G` column-wise and and checks for each entry whether there is an edge in either direction between the corresponding vertices. Your job is to devise a conversion routine that is as fast as possible using the techniques we saw in class (e.g., loop unrolling and blocking).

Table 1 and 2 summarize the performance of the naive implementation in terms of the cache behavior and the throughput for 5 different values of N. We evaluate the cache performance of a conversion routine by looking at the total number of cache misses compared to the size of the matrix. More precisely, we use the *miss score* of the routine to measure the cache performance, where the miss score is defined as $\#\texttt{misses}/(N \times N)$. Since the miss score is directly proportional to the total number of misses, the cache behavior of an implementation is the better the *lower* the miss score is. The throughput is measured in megabytes per second on one of our Pentium III Xeon machines. That is, if converting an $N \times N$ adjacency matrix requires T seconds, then the throughput is $(N \times N \times 4 \times 10^{-6})/T$.

If you look at Table 1, you'll find that the naive implementation has an average miss score of 0.47 which is quite high. In particular, for the larger graphs the miss score is very high. Therefore, a good point to start is to try to reduce the number of cache misses. This is exactly what you are asked to do in part 1. Then, in part 2 of the lab you will focus solely on maximizing the throughput.

To make life easier, you can assume that N is a multiple of 32. Your code must run correctly for all such values of N, but we will measure its performance only for the 5 values shown in Table 1. To summarize the overall effect of your optimizations, we will compute the *geometric mean* of the results for these 5 values,

Test case	1	2	3	4	5	
Method N	32	64	128	256	1024	Geom. Mean
Naive	0.1269	0.1345	1.1076	1.1198	1.1238	0.4735
Performance we were able to achieve in our solutions to Parts I and II:						
Good Cache (Part I)	0.1279	0.1259	0.1253	0.1250	0.1628	0.1327
Good Throughput (Part II)	0.1279	0.1262	0.1254	0.1251	0.1767	0.1349

Table 1: Simulated Cache Miss Scores for Conversion Routines

i.e. if the measured results (throughputs or miss scores) are R_{32}, R_{64}, R_{128}, R_{256}, and R_{1024} then we compute the overall performance as

$$R = \sqrt[5]{R_{32} \times R_{64} \times R_{128} \times R_{256} \times R_{1024}}$$

Part I: Optimizing Simulated Cache Performance

A good way to improve the performance of the implementation is to first focus on its cache behavior. To measure the cache behavior 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 in Part II to, so a good implementation here will serve you well when optimizing throughput.

We instrument the implementation of the conversion routine 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. As mentioned above, we will measure the cache performance in terms of the *miss score* which is defined as `#misses`$/(N \times N)$.

Your task for this part of the assignment is to modify the code for the routine named `good_cache` in `conversion.c` to achieve as low a miss score as you can. We provide you with a driver program that generates test graphs for the graph sizes in Table 1 and runs your routine `good_cache` on it. To check your progress first compile the code on your machine with the command `make driver`. After that, you can use two scripts to test your solution. The first one is `driver_one_test` which runs the driver program for a specific test case, i.e. a specific matrix size. For example, to run your solution for $N = 64$ vertices, you would type:

```
% driver_one_test 2
```

The program will output cache performance statistics (and throughput results that are pertinent to Part II) from the cache simulator (the total number of misses, and the number of read and write misses) and the miss score, which will be used to to measure the cache performance of your program. You can also run all the test cases in once using `driver_all`. Table 1 shows the improvement our implementation of `good_cache` was able to obtain. As you can see, for the larger matrices it is possible to improve upon the naive implementation by almost a factor of 10!

Test case	1	2	3	4	5	
Method	32	64	128	256	1024	Geom. Mean
Naive	72.3	79.3	51.5	41.36	19.4	47.3
Performance we were able to achieve in our solutions to Parts I and II:						
Good Cache	142.9	143.3	137.9	136.16	114.5	134.5
Good Throughput	139.5	176.8	172.8	174.62	142.1	160.2

Table 2: Measured Throughput of Conversion Routines (MB/s)

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

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

and commenting out the lines:

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

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

You may also want to disable cacheprof once you are done with Part I, so you don't have to wait for the cache simulation to complete when working on Part II. You can do so by commenting out the following lines in the `Makefile`:

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

and running `make clean; make driver`.

Part II: Optimizing Measured Throughput

Your task for second part of this assignment is to write a routine `good_throughput` that has as high a throughput as possible. Your grade will depend on the geometric mean of the throughputs you achieve across the 5 values of N. Improving the actual performance of your routine requires a fair amount of low-level code optimization. The best method is to repeatedly transform the code, and check the generated assembly to see how much simplification you've obtained, and benchmark the code. You may find it helpful to also look at the cache performance for `good_throughput`. You can do so by un-commenting the following lines in the `Makefile`:

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

and using `make clean; make driver`. `cacheprof` will then instrument `good_throughput`, allowing you to examine its cache performance as you optimize it. Remember that cache simulator 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 more accurate results, you can submit your implementation to the class server. Please look at the WWW page of this lab for how to do this.

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 not interfere with the cache simulation or time measurement mechanism. You will also be penalized if your code prints any extraneous information.

You are allowed to define macros, additional global variables, and other procedures in the files `conversion.c` and `conversion_through.c`. If you modify any other files, make sure your `conversion.c` and `conversion_through.c` still compile and run with the original versions of these files.

Evaluation

We will use the performance of the "Good Cache" and "Good Throughput" routines given in Table 1 and Table 2 as a reference solution.

Your grade will be based on the following:

- Correctness: You will get NO CREDIT for buggy code! This includes code that correctly converts the provided test matrices, but incorrectly converts other matrices.
- Cache performance: You will get full credit for Part I if your implementation is correct and achieves a mean miss score of at most 0.1327. 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 160.2. You will get partial credit for a correct implementation that does better than the supplied naive one.

Hand In

Please see the WWW page of this lab for how to hand it your solutions to the class server.