CSE 6230 - Spring 2024
Instructor: Prof. Helen Xu

Homework 1 - Tools and Cache-friendly programming

Due: Jan 22 @ 4:55pm on Canvas
Deliverables: Code and writeup

This homework is a hands-on exercise that will try several of the coding tools available on PACE ICE, the Georgia Tech instructional cluster. We will be using the tools to study dense matrix multiplication. You should use this assignment to become familiar with the tools that we will be using throughout the course.

For this homework, we will be focusing on optimizing and using tools on **serial code**. Do not use multithreading or multiple nodes.

Your homework should be completed **individually** (not in groups).

Code and some parts of the exercises are from MIT 6.172.

Setup

This section will explain how to get onto the computing environment, download, and run the code.

Log into PACE ICE

Please check this Getting Started with ICE guide, which has useful information about how to use the instructional cluster.

The first step is to log on, which you should be able to do since you are registered for this course. If there are any issues, please contact the course staff.

You will also need to be connected to Georgia Tech's VPN to use either ssh or OnDemand. Connect to the client-based GlobalProtect VPN before attempting to access PACE resources.

From the terminal

If you are familiar with computing terminals (e.g., Windows Powershell or Mac terminal), you can use it to access the cluster via SSH to **`login-ice.pace.gatech.edu`**. Connect with your GT username and password.

From the terminal, enter `ssh <your-username-here>@login-ice.pace.gatech.edu`, then provide your GT password at the prompt. No asterisks will appear, so enter your password then `<Enter>`.

For example, `ssh hxu615@login-ice.pace.gatech.edu`.

From the web-based interface

You can also access ICE through a web browser (requires VPN connection as described above).

Access it here: https://ondemand-ice.pace.gatech.edu/

Getting the code

Clone from the git repo:

```
$ git clone https://github.com/cse6230-spring24/hw1.git
```

Homework submission instructions

There are two parts to the homework: a writeup and the code. The writeup should be in pdf form, and the code should be in zip form. You can submit multiple files - **you should submit 2** - by choosing "+ Add Another File" in Canvas under "Homework 1."

To get the code from PACE ICE to your local machine to submit, first zip it up on PACE ICE:

```
$ zip hw1.zip hw1/
```

Then from the terminal (or terminal equivalent) your local machine (e.g., your personal laptop), scp the code over.

```
$ scp <your-username-here>@login-ice.pace.gatech.edu:<path-to-hw1.zip>
<local-file-location>
```

For example, if I had `hw1.zip` in my home directory on PACE ICE and wanted to copy it into my current directory on my local machine, I would run:

```
$ scp hxu615@login-ice.pace.gatech.edu:~/hw1.zip .
```

Note: There is not a lot of space available on the home directories on PACE ICE. Each Pace-ICE user is also provided with a local scratch directory (at `~/scratch`) where they should store their files.

Cluster basics (from PACE ICE documentation)

There are two types of machines on PACE ICE: **head nodes** and **compute nodes**.

Head nodes

- Where you log in
- Submit jobs from here
- Can edit and compile small-scale programs
- Access storage
- Login nodes are shared by all. Please do not use the login nodes for any resource-intensive activities, as it prevents other students from using the cluster. PACE will stop processes that continue for too long or use too many resources, in order to ensure functionality of the login node. Please use a compute node for all computational work. An interactive job provides an interactive environment on a compute node for debugging and other resource-intensive activities.

Compute nodes

- For running all computations
- Assigned by the scheduler and accessed only when assigned
- Access storage
- May vary in their computational capability

To complete the homework, you probably want to use an interactive job, which allows interactive use so you can work “live” on a compute node. Here is the documentation for how to start one of these jobs.

For example, the following command uses `salloc` to allocate 1 node with 4 cores for 1 hour for an interactive job:

```
$ salloc -N1 --ntasks-per-node=4 -t1:00:00
```

Since this homework focuses on single-node and single-threaded code, you only need to ask for **one node and 1 task per node**.

Part 0: Building and running the code

Part 0 and 1 of this homework requires gcc/10.3.0-o57x6h, which is loaded by default on PACE.

To compile the code, go into the homework directory and type make. That is, from the directory where you cloned the homework, type:

```
$ cd hw1
$ make
```

You should see some output like:

```
gcc -O1 -DNDEBUG -Wall -std=c99 -D_POSIX_C_SOURCE=200809L -c testbed.c
-o testbed.o
gcc -O1 -DNDEBUG -Wall -std=c99 -D_POSIX_C_SOURCE=200809L -c
matrix_multiply.c -o matrix_multiply.o
gcc -o matrix_multiply testbed.o matrix_multiply.o
```

Notice that we are compiling with optimization level 1 (i.e., `-O1`).

Exercise: Modify the Makefile so that the program is compiled using optimization level 3 (i.e., `-O3`).

Write-up 1: Now, what do you see when you type `make clean; make`?

You can then run the built binary by typing `./matrix_multiply`. The program should print out something then crash with a segmentation fault.

Part 1: Debugging

Using GDB

While running your program, if you encounter a segmentation fault, bus error, or assertion failure, or if you just want to set a breakpoint, you can use the debugging tool GDB.

Exercise: Start a debugging session in GDB:

```
$ gdb -args ./matrix_multiply
```

This command should give you a GDB prompt, at which you should type `run` or `r`:

```
$ (gdb) run
```

Your program will crash, giving you back a prompt, where you can type `backtrace` or `bt` to get a stack trace:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555904 in matrix_multiply_run ()
(gdb) bt
#0  0x0000555555555904 in matrix_multiply_run ()
#1  0x00005555555553f7 in main ()
```

(You may see an error like “Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.x86_64” - it is ok to ignore the error.)

This stack trace says that the program crashes in `matrix_multiply_run`, but doesn’t tell any other information about the error. In order to get more information, build a “debug” version of the code. First, quit GDB by typing `quit` or `q`. Next, build a “debug” version of the code by typing `make DEBUG=1`:

```
make clean; make DEBUG=1
rm -f testbed.o matrix_multiply.o matrix_multiply .buildmode \
        testbed.gcda matrix_multiply.gcda \
        testbed.gcno matrix_multiply.gcno \
        testbed.c.gcov matrix_multiply.c.gcov fasttime.h.gcov
gcc -g -DDEBUG -O0 -Wall -std=c99 -D_POSIX_C_SOURCE=200809L -c testbed.c
-o testbed.o
gcc -g -DDEBUG -O0 -Wall -std=c99 -D_POSIX_C_SOURCE=200809L -c
matrix_multiply.c -o matrix_multiply.o
gcc -o matrix_multiply testbed.o matrix_multiply.o
```

The major differences from the optimized build are '`-g`' (add debug symbols to your program) and '`-O0`' (compile without any optimizations). Once you have created a debug build, you can start a debugging session again:

```
(gdb) r
Starting program: /home/helen/hw1-testing/matrix_multiply
Setup
Running matrix_multiply_run()...

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555c8d in matrix_multiply_run (A=0x555555559360,
B=0x5555555592c0,
    C=0x555555559460) at matrix_multiply.c:90
90                C->values[i][j] += A->values[i][k] * B->values[k][j];
```

Now, GDB can tell that a segmentation fault occurs at `matrix_multiply.c` line 90. You can ask GDB to print values using `print` or `p`:

```
(gdb) p A->values[i][k]
$1 = 7
(gdb) p B->values[k][j]
Cannot access memory at address 0x0
(gdb) p B->values[k]
$2 = (int *) 0x0
(gdb) p k
$3 = 4
```

This suggests that `B->values[4]` is `0x0`, which means B doesn't have row 5. There is something wrong with the matrix dimensions.

Using assertions

The `tbassert` package is a useful tool for catching bugs before your program goes off into the weeds. If you look at `matrix_multiply.c`, you should see some assertions in `matrix_multiply_run` routine that check that the matrices have compatible dimensions.

Exercise: Uncomment these lines and a line to include `tbassert.h` at the top of the file. Then, build and run the program again using GDB. Make sure that you build using `make DEBUG=1`. You should see:

```
(gdb) r
...
Running matrix_multiply_run()...
matrix_multiply.c:78 (matrix_multiply_run) Assertion A->cols == B->rows
failed: A->cols = 5, B->rows = 4

Program received signal SIGABRT, Aborted.
__GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:50
50      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
```

Now, GDB says that `“Assertion ‘A->cols == B->rows’ failed”`, which is much better than the former segmentation fault. The assertion provides a printf-like API that allows you to print values in your own output, as above. However, even if you don't print values in your assertions, the debug build still has the symbols for GDB, as above. Unlike the above, however, if you try to print `A->cols`, you will fail. The reason is that GDB is not in the stack frame you want. You can get the stack trace to see which frame you want (#3 in this case), and type `frame 3` or `f 3` to move to frame #3. After that, you can print `A->cols` and `B->cols`.

```
(gdb) bt
#0  __GI_raise (sig=sig@entry=6) at
../sysdeps/unix/sysv/linux/raise.c:50
#1  0x00007ffff7de0859 in __GI_abort () at abort.c:79
#2  0x0000555555555ca6 in matrix_multiply_run (A=0x555555559360,
B=0x5555555592c0,
    C=0x555555559460) at matrix_multiply.c:78
#3  0x0000555555555964 in main (argc=1, argv=0x7fffffffe3e8) at
testbed.c:133
(gdb) f 3
#3  0x0000555555555964 in main (argc=1, argv=0x7fffffffe3e8) at
testbed.c:133
133       matrix_multiply_run(A, B, C);
(gdb) p A->cols
$1 = 5
(gdb) p B->rows
$2 = 4
```

You should see the values 5 and 4, which indicates that we are multiplying matrices of incompatible dimensions.

You will also see an assertion failure with a line number for the failing assertion without using GDB. Since the extra checks performed by assertions can be expensive, they are disabled for optimized builds, which are the default in our Makefile. As a result, if you make the program without `DEBUG=1`, you will not see an assertion failure.

You should consider sprinkling assertions throughout your code to check important invariants in your program, since they will make your life easier when debugging. In particular, most nontrivial loops and recursive functions should have an assertion of the loop or recursion invariant.

Exercise: Fix `testbed.c`, which creates the matrices, rebuild your program, and verify that it now works. You should see “Elapsed execution time...” after running

```
$ ./matrix_multiply
```

Next, check the result of the multiplication. Run

```
$ ./matrix_multiply -p
```

The program will print out the result. The result seems to be wrong, however. You can check the multiplication of zero matrices by running

```
$ ./matrix_multiply -pz
```

Using a memory checker

Some memory bugs do not crash the program, so GDB cannot tell you where the bug is. You can use the memory checking tool Valgrind to track these bugs.

Valgrind won’t be initially loaded in the environment when you’re running an interactive job on the PACE cluster. To load the module for your environment, run:

```
$ module load valgrind
```

Then run:

```
$ make clean && make DEBUG=1
```

to get rid of the existing build and get a fresh build. Then run Valgrind using

```
$ valgrind ./matrix_multiply -p
```

You need the `-p` switch, since Valgrind only detects memory bugs that affect outputs. You should also use a “debug” version to get a good result. This command should print out many lines. The important ones are:

```
==1487776== Use of uninitialised value of size 8
==1487776==    at 0x48BE69B: _itoa_word (_itoa.c:179)
==1487776==    by 0x48DA574: __vfprintf_internal
(vfprintf-internal.c:1687)
==1487776==    by 0x48C4D3E: printf (printf.c:33)
==1487776==    by 0x109BE6: print_matrix (matrix_multiply.c:68)
==1487776==    by 0x1099A3: main (testbed.c:139)
```

This output indicates that the program used a value before initializing it. The stack trace indicates that the bug occurs in `testbed.c:139`, which is where the program prints out matrix C.

Exercise: Fix `matrix_multiply.c` to initialize values in matrix C before using them. Keep in mind that the matrices are stored in structs. Rebuild your program, and verify that it outputs a correct answer.

Write-up 2: After you fix your program, run `./matrix_multiply -p` and `./matrix_multiply -pz`. Paste the program output showing that the matrix multiplication is working correctly.

Memory management

The C programming language requires you to free memory after you are done using it, or else you will have a memory leak. Valgrind can track memory leaks in the program. Run the same Valgrind command, and you will see these lines at the very end:

```
==1487855== LEAK SUMMARY:
==1487855==    definitely lost: 48 bytes in 3 blocks
==1487855==    indirectly lost: 288 bytes in 15 blocks
==1487855==      possibly lost: 0 bytes in 0 blocks
==1487855==    still reachable: 0 bytes in 0 blocks
==1487855==         suppressed: 0 bytes in 0 blocks
```

This output suggests that there are indeed memory leaks in the program. To get more information, you can build your program in debug mode and again run Valgrind, using the flag `--leak-check=full`:

```
$ valgrind --leak-check=full ./matrix_multiply -p
```

The trace shows that all leaks are from the creations of matrices A, B, and C.

Exercise: Fix `testbed.c` by freeing these matrices after use with the function `free_matrix`. Rebuild your program, and verify that Valgrind doesn't complain about anything.

Write-up 3: Paste the output from Valgrind showing that there is no error in your program.

Part 2: Profiling

Callgrind is a profiling tool under Valgrind that can construct a call graph for a program's run. By default, the collected data consists of the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the numbers of such calls. Optionally, a cache simulator can produce further information about the memory access behavior of the application.

The profile data is written out to a file at program termination which can be later used for presentation of the data, and interactive control of the profiling.

To use callgrind, pass it in as one of the parameters to Valgrind:

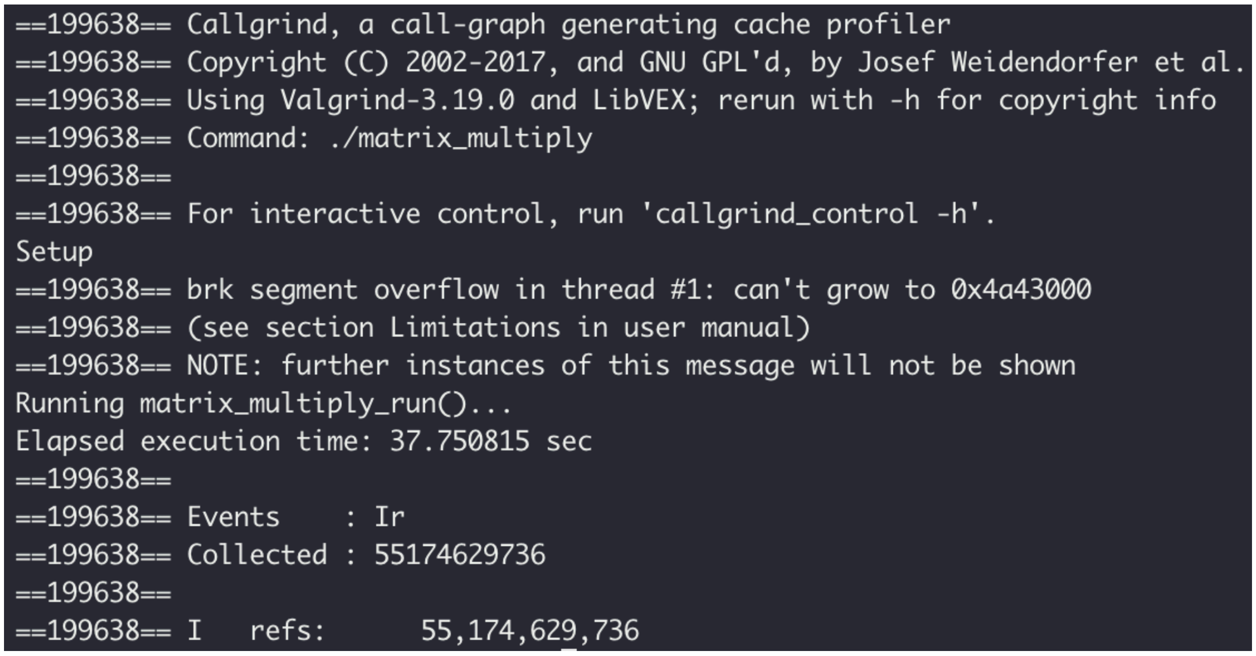
```
$ valgrind --tool=callgrind [callgrind options] your-program [program options]
```

Note: To get accurate information about the line numbers, function names, file names, etc. through callgrind, the executable must be **compiled with debug symbols**.

For example, to run it on our matrix multiply code, run:

```
$ make clean && make DEBUG=1
$ valgrind --tool=callgrind ./matrix_multiply
```

You should see an output like this:



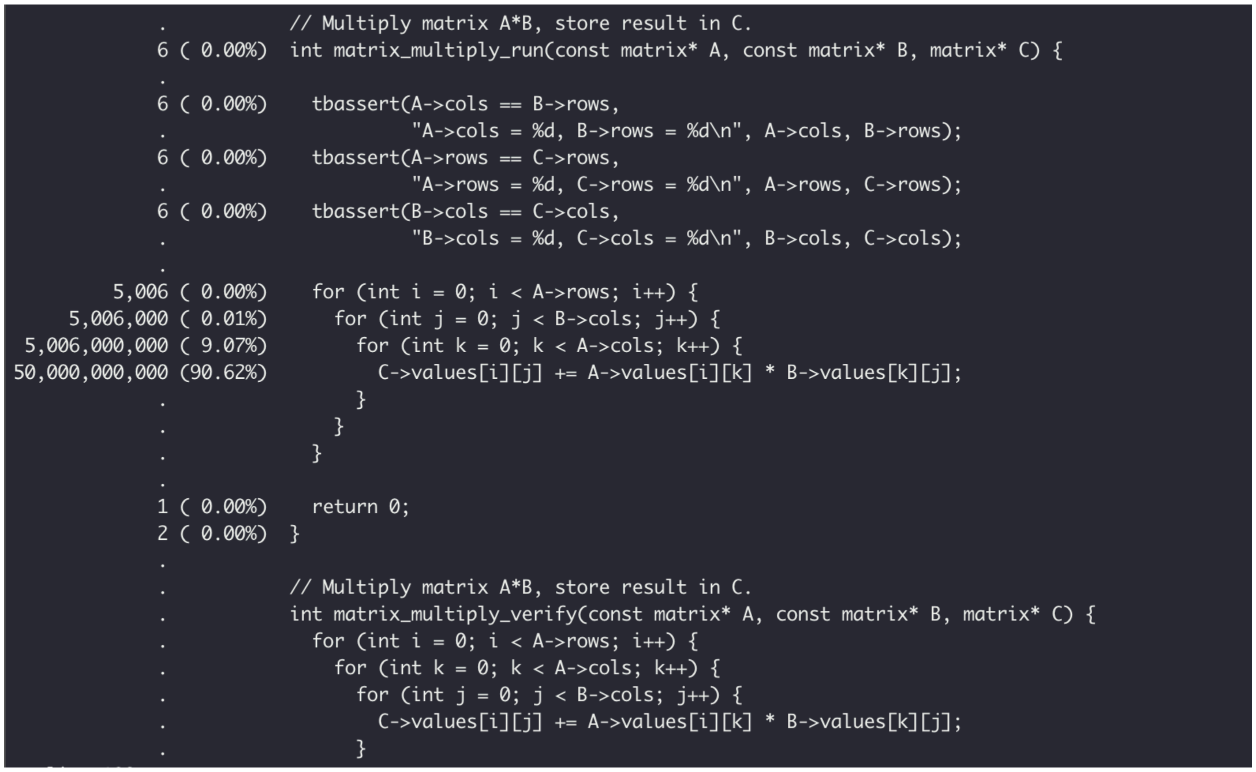
```
==199638== Callgrind, a call-graph generating cache profiler
==199638== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==199638== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==199638== Command: ./matrix_multiply
==199638==
==199638== For interactive control, run 'callgrind_control -h'.
Setup
==199638== brk segment overflow in thread #1: can't grow to 0x4a43000
==199638== (see section Limitations in user manual)
==199638== NOTE: further instances of this message will not be shown
Running matrix_multiply_run()...
Elapsed execution time: 37.750815 sec
==199638==
==199638== Events    : Ir
==199638== Collected : 55174629736
==199638==
==199638== I   refs:      55,174,629,736
```

After program termination, a profile data file named `callgrind.out.<pid>` is generated, where pid is the process ID of the program being profiled. The data file contains information about the calls made in the program among the functions executed, together with events of type Instruction Read Accesses (Ir).

To generate a function-by-function summary from the profile data file, we will use `callgrind_annotate` which reads in the profile data and, and prints a sorted lists of functions, optionally with source annotation.

The partial output around `run_matrix_multiply` should look similar to this:

```
$ callgrind_annotate callgrind.out.<pid>
```



```
             .           // Multiply matrix A*B, store result in C.
             6 ( 0.00%)  int matrix_multiply_run(const matrix* A, const matrix* B, matrix* C) {
             .
             6 ( 0.00%)    tbassert(A->cols == B->rows,
             .                      "A->cols = %d, B->rows = %d\n", A->cols, B->rows);
             6 ( 0.00%)    tbassert(A->rows == C->rows,
             .                      "A->rows = %d, C->rows = %d\n", A->rows, C->rows);
             6 ( 0.00%)    tbassert(B->cols == C->cols,
             .                      "B->cols = %d, C->cols = %d\n", B->cols, C->cols);
             .
         5,006 ( 0.00%)    for (int i = 0; i < A->rows; i++) {
     5,006,000 ( 0.01%)      for (int j = 0; j < B->cols; j++) {
 5,006,000,000 ( 9.07%)        for (int k = 0; k < A->cols; k++) {
50,000,000,000 (90.62%)          C->values[i][j] += A->values[i][k] * B->values[k][j];
             .                 }
             .               }
             .             }
             .
             1 ( 0.00%)    return 0;
             2 ( 0.00%)  }
             .
             .           // Multiply matrix A*B, store result in C.
             .           int matrix_multiply_verify(const matrix* A, const matrix* B, matrix* C) {
             .             for (int i = 0; i < A->rows; i++) {
             .               for (int k = 0; k < A->cols; k++) {
             .                 for (int j = 0; j < B->cols; j++) {
             .                   C->values[i][j] += A->values[i][k] * B->values[k][j];
             .                 }
```

Exercise: In `testbed.c`, increase the size of the matrix dimension (`kMatrixSize`) to 1000. This will make the program multiply matrices of size 1000x1000. You can also experiment with other matrix dimensions.

Write-up 4: Run `./matrix_multiply` under Callgrind with 1000x1000 matrices. Annotate the results using `callgrind_annotate`. Does the event counts change as expected? Briefly, discuss how we can utilize the event counts and percentages to optimize our codes.

Optional: For graphical visualization of the data, you can try KCachegrind, which is a KDE/Qt based GUI that makes it easy to navigate the large amount of data that Callgrind produces.

Other Profiling Tools. Choosing a code profiling tool depends on the task's needs, personal preference, and software availability. Here is a list of popular tools:

- Intel Vtune Profiler – See these instructions to use vtune on pace-ice.
- Perf

- Nvidia Nsight for GPUs

Part 3: Cachegrind

Cachegrind (a Valgrind tool) is a cache and branch-prediction profiler. Recall from class that a read from the L1 cache can be around 100x faster than a read from RAM! Optimizing for cache hits is a critical part of performance engineering.

Cachegrind simulates how your program interacts with a machine's cache hierarchy and branch predictor and can be used even in the absence of available hardware performance counters.

Here is an example on how to identify cache misses, branch misses, clock cycles, and instructions executed by your program using Cachegrind:

```
$ valgrind --tool=cachegrind --branch-sim=yes <program_name>
<program_arguments>
```

For example, to run it on our matrix multiply code, run:

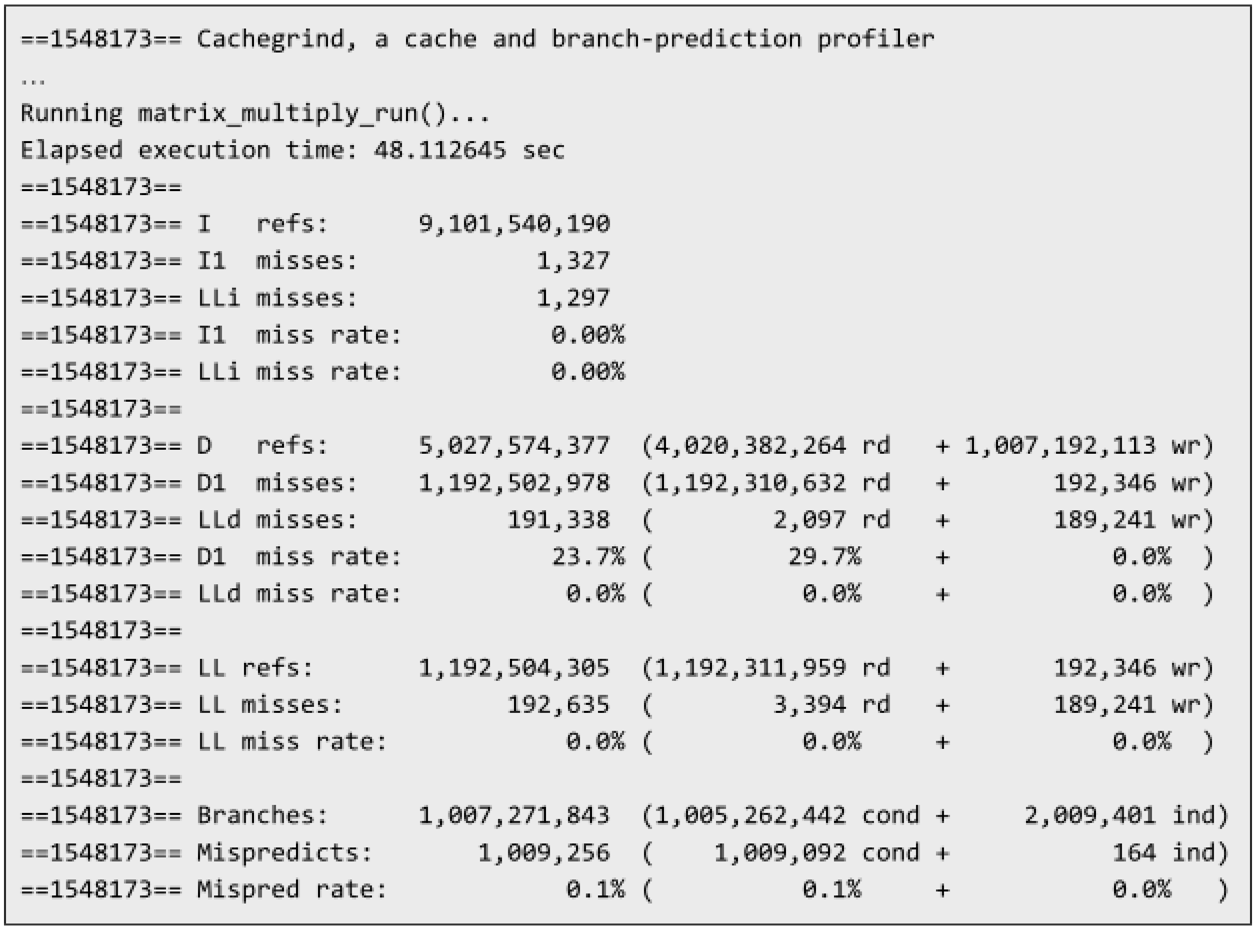
```
$ valgrind --tool=cachegrind --branch-sim=yes ./matrix_multiply
```

Note: Although `valgrind --tool=cachegrind` measures cache and branch predictor behavior using a simulator, it bases its simulation upon the architecture on which it is run. You should expect different results when running on different machines.

Exercise: In `testbed.c`, verify that the size of the matrix dimension (`kMatrixSize`) is 1000 (you should have changed this in Part 2).

Now, run `make clean; make` to rebuild it, and then run with Cachegrind.

You should see output like the following:



```
==1548173== Cachegrind, a cache and branch-prediction profiler
...
Running matrix_multiply_run()...
Elapsed execution time: 48.112645 sec
==1548173==
==1548173== I   refs:      9,101,540,190
==1548173== I1  misses:            1,327
==1548173== LLi misses:            1,297
==1548173== I1  miss rate:          0.00%
==1548173== LLi miss rate:          0.00%
==1548173==
==1548173== D   refs:      5,027,574,377  (4,020,382,264 rd   + 1,007,192,113 wr)
==1548173== D1  misses:    1,192,502,978  (1,192,310,632 rd   +       192,346 wr)
==1548173== LLd misses:          191,338  (        2,097 rd   +       189,241 wr)
==1548173== D1  miss rate:          23.7% (         29.7%     +           0.0%  )
==1548173== LLd miss rate:           0.0% (          0.0%     +           0.0%  )
==1548173==
==1548173== LL refs:       1,192,504,305  (1,192,311,959 rd   +       192,346 wr)
==1548173== LL misses:           192,635  (        3,394 rd   +       189,241 wr)
==1548173== LL miss rate:            0.0% (          0.0%     +           0.0%  )
==1548173==
==1548173== Branches:      1,007,271,843  (1,005,262,442 cond +     2,009,401 ind)
==1548173== Mispredicts:       1,009,256  (    1,009,092 cond +           164 ind)
==1548173== Mispred rate:            0.1% (          0.1%     +           0.0%   )
```

Write-up 5: Run `./matrix_multiply` under Cachegrind to identify cache performance. It may take a little while. In the output, look at the D1 and LLd misses. D1 represents the lowest-level cache (L1), and LL represents the last (highest) level data cache (on most machines, L3). Do these numbers correspond with what you would expect? Try playing around with the matrix dimension. How can you bring down the number of cache misses?

Part 4: Performance optimizations

We want to identify performance bottlenecks and incrementally improve the performance of the matrix multiply as explained during lecture and through the tasks below.

Note: Make sure you're evaluating the non-debug version for the rest of your experiments.

Exercise: We should have increased the size of the matrices to 1000x1000 in the previous section. If you have not done so, do so now.

Now let's try one of the techniques from the first lecture. Right now, the inner loop produces a sequential access pattern on `A` and skips through memory on `B`.

Let's rearrange the loops to produce a better access pattern.

Exercise: First, you should run the program as is to get a performance measurement. Next, swap the `j` and `k` loops in `matrix_multiply_run` in `matrix_multiply.c`, so that the inner loop strides sequentially through the rows of the `C` and `B` matrices. Rerun the program, and verify that you have produced a speedup.

Also, you can run with -v to verify that the result is still correct. That is, run

```
$ ./matrix_multiply -v
```

You should see output like this:

```
Setup
Running matrix_multiply_run()...
Verified!
Elapsed execution time: 1.924505 sec
```

Note: Runtimes and Cachegrind results should be measured without -v.

Write-up 6: Report the execution time of your matrix multiply before and after the optimization. Also, paste the output from Cachegrind after the loop interchange. What differences do you notice between the two Cachegrind outputs (before and after the optimization)?

Exercise: Next, implement one level of matrix blocking/tiling as discussed in lecture. Be sure to verify that the result is still correct.

What is the best tile size? Try several sizes and report the resulting runtimes in the writeup.

Write-up 7: Report the execution time of your matrix multiply with tile sizes of 4, 8, 16, 32, 64, and 128. Also, paste the output from Cachegrind for the fastest tile size you found. What differences do you notice between the two Cachegrind outputs (before and after tiling)?

That's the end of this homework! Submit your writeup and code as described in the "Homework submission instructions" above.

+