Long Latency Operations

Daniel Jiménez-González

March 6, 2018

Index

1

Previous Work

We provide you a *.zip* file with several input files.

- There are several Makefile targets to automatically generate the binary files. For instance, *make prog.pg* will generate an executable compiled with profiling flags. Using *CFLAGS* variable in the Makefile in a subdirectory (local Makefile) of a program you can define the specific compilation flags for a program. Using *PROGS* variable in the local Makefile you can do "make" of different program names. You can read README file in the root directory of the laboratory lab3.
- Most of the programs have a parameter in order to modify the amount of work to do. You can modify that parameter in order to obtain a reasonable execution time.
- Most of the programs use a random input data. However, as they don't modify the seed of the random function, they should obtain the same results for different executions.

Some advices for the lab activities:

- You must redirect the output of the program executions to a file since the output can be large and/or with not printable characters.
- To compare ASCII output files, you should use tools like *diff*, *tkdiff* or *cmp*.
- To compare binary output files, you should use *cmp* tool.
- In order to visualize a file with non printable characters you can use the od tool (octal dump). For instance, `od -b /bin/ls | more` shows the ASCII code of the characters of the file `/bin/ls`. That tool may be parametrized with the starting point to do the `dump`.

In order to take profit of the activities at the laboratory you should do the following activities before going to the lab:

1.1 Arithmetic Expression Optimizations

1. We give you the `primers.c` program that uses the Eratostenes method in order to create a prime list (from 1 upto the number you indicate as parameter).
    - Look at http://www.wikipedia.org and the GCC low-level runtime library for information.
    - Try to find something regarding the `__udivdi3` and `__umoddi3`. That will help you with laboratory exercises. Hint: those symbols do not appear depending on the architecture you compile to.

1.2 Memoization

2. Look for information regarding to the trigonometric routine implementations `sin()` and `cos()`. Is there any hardware instruction to do it in the Cortex A9 of our boards?

2

Arithmetic Expression Optimizations

1. `primers.c` program uses the Eratostenes method in order to create a prime list (from 1 upto the number you indicate as parameter).

   (a) Compile `primers` with "-O0" (`make primers.0` and do timing, and keep the original output result.

   (b) Use profiling (with `gprof`,`valgrind` and/or `operf, opannotate`) to analyze its behavior when compiling with -O0. Note: you may want to use the primers argument to reduce the computation time with `valgrind`.

   (c) Which are the most time consuming functions? Look at their most time consuming lines to figure out which are the expensive operations. Is there any way to reduce/avoid those costy operations?

   (d) Copy the original code to primers_opt.c. Optimize this copy of primers based on the analysis done in the previous question.

   (e) Compile it using `make primers_opt.g` and check that the results of the optimized version you have done are the same than the original one.

   (f) Which speedup have you obtained with your optimized code compared to the original `primers` when compiling both of them with "-O0"?

   (g) Now, compile the original and the optimized versions of `primers` with "-O3" and do timing of both of them.

   (h) Which speedup have you obtained compared to the original `primers` when compiling with "-O3" both programs, original and optimized? Do profiling again and look a the output of `objdump -d` in order to explain the speedup obtained.

3

Memoization and Buffering

2. `trigon.c` program performs several calls to write and the trigonometric routines `sin()` and `cos()`.
   (a) Study the source code of the program.
   (b) Compile the program with `make trigon.pg3`. Profile and analyze the program behavior. Explain the results.
   (c) Re-compile the program with `make trigon.pg3s`. Profile and analyze the program behavior. Explain the results.
   (d) Figure out how many system calls are done in the program, and which is the elapsed time of the program. Note that `strace` may take a while to find out this information.
   (e) Based on the previous analysis, modify the program in order to avoid so many calls and reduce the overall elapsed time.
   (f) Compute the speed-up of your optimized version compared to the original version.
   (g) Profile the new code. How much CPU time is devoted to trigonometric computations? Which is the maximum speedup that we can achieve if we improve the corresponding execution time?
   (h) Modify the program in order to avoid the computation repetitions and check that the output of the new version is the same as the original code.
   (i) Profile the new version of the program and compare it with the profile of the original code.
   (j) Which overall speed-up have you obtained?

4

Routine Specialization

3. `pi.c` program compute the first 10000 decimals of the π number.

   (a) What is the speedup of the `pi` program compiled with `O3` compared to the `pi` compiled with `O0`?

   (b) Profile both `pi` binaries (compiled with `O3` and with `O0`) using `valgrind` and `callgrind_annotate`. Open the assembler code to understand the differences (`objdump -d`) and/or look for the places where "divide instrucions" are executed using `callgrind_annotate` (Note: use `1000` as argument of `pi.c` when using valgrind). Could you justify these differences?

   (c) It seems that the compiler has done a good work but there is still work to do. Based on the previous profiling, look at the source code of the program, and in particular, the source code of routine `DIVIDE`. Then, propose an optimization that can help to reduce the cost of each `specific` call to DIVIDE in order to reduce/avoid long latency operations. Specilize the code of pi using memoization. Some hints:

      - For each DIVIDE by a particular value, `x[k]`, `r`, and `u` in the code will always have the same range of values (maybe different range among them)
      - Those values in the range are for a limited number of input values

   (d) Profile the new version of the program and compare it with the profile of the original code.

   (e) Which speed-up have you obtained?