Computer Architecture Fall 2023
Assignment 3: Optimizing Single-Thread Performance

Deadline: Friday, December 1, 2023

In this lab assignment, we'll investigate how to improve the performance of single-threaded programs. There are several ways how the run time of a program can be reduced. This assignment will focus on reducing memory latency, by making better use of the cache and reducing the amount of memory accesses.

To this end, four benchmark programs are provided on Brightspace. It will be your task to optimize each of the four programs using a different technique. Using the tools `cachegrind` and `perf`, you'll have to conduct experiments to determine how each technique affects the performance, and document the results in a report. In the report, you should not just report the raw numbers obtained from the tools. Rather, we expect an explanation of *what* was measured and *why*, *how* it was measured, as well as an explanation and discussion of the results.

You should also submit *all variants* of your optimized programs that were used for the experiments, so that your results can be reproduced. We will judge the programs on correctness, but not on code quality.

Cache Performance Measurement

Each of the four benchmark programs encapsulates a certain computation that we would like to study. In real-world programs, most of the run time is spent in such computations ("hot spots"). This kind of core computation is often referred to as a "computational kernel".

There are two tools that you can use for measuring a program's cache performance: `cachegrind`, and `perf`. A document explaining how to use these programs is available on Brightspace. In some cases, one run of the kernel on the real hardware may only take a few milliseconds to execute. Therefore, when performing experiments using `perf`, a kernel is usually repeated several times (50 – 500) for more accurate measurement. The benchmark programs already provide a means to do this.

You are not allowed to change the compiler flags in the provided `Makefile`: these flags are set to avoid compiler optimizations accidentally affecting the results measured from your experiments.

⚠ Do not use the SSH servers (`huisuil` or `remotelx`) for experiments, but connect to a regular computer in the computer lab instead. The SSH server hardware may be virtualized or use large L3 caches, leading to unexpected results.

Task 1: Grayscale

The first problem, `grayscale.c`, is a program that takes a PNG image and converts it into grayscale. To load images a small support library (in `support/image.c`) is used. Each pixel in the image is stored as a 32-bit integer, wherein each color channel (red, green, blue, and alpha) is

stored using 8 bits (`RGB8888` format). The macro `RGBA_unpack` extracts the individual channels from the 32-bit value, and converts them to separate floats from 0.0 to 1.0. The macro `RGBA_pack` does the reverse. The pixels themselves are stored as a one-dimensional array `data`, in row-major order. The property `rowstride` indicates how many bytes apart one row is from the next. The macro `image_get_pixel` can be used to determine the address of a pixel given its (x, y) coordinates.

Images that will be useful for your experiments can be found in the following directory: `/vol/share/groups/liacs/scratch/ca2023/lab3data/`.

Currently, `grayscale.c` contains a function `grayscale_xy`, which processes all pixels by first looping over the X coordinate, and then over the Y coordinates. You task is to **investigate the effectiveness of the loop interchange transformation for this particular kernel**.

Make a copy of the function, `grayscale_yx`, where you apply the loop interchange. Add a new target with the same name to the list `TARGETS` in the `Makefile`. Now, running `make` will build both variants. Note that the program outputs the results in CSV format to help you collect and organize experimental results for further processing.

In your report, investigate the following three research questions. Use *multiple* images in your experiments, and remember to explain the results: why is there an improvement? Does it match your expectations?

RQ1. Show the effectiveness by presenting relevant cache performance statistics from cachegrind.

RQ2. Compare the execution time of the two variants.

RQ3. Show the effectiveness by presenting cache performance on real hardware collected by `perf`.

Task 2: Matrix multiplication

The second problem, `matrixmul.c`, performs a matrix multiplication. **Apply loop blocking and investigate its performance**. An explanation of loop blocking is given in the lecture slides; a more in-depth explanation is also available on Brightspace. Make a copy of the file `matrixmul.c` beforehand, so you can compare to the version without loop blocking.

⚠ Do not use `fmin()` to implement the `min` needed in the loop condition. Using this function will introduce a function call for each loop iteration as well as perform integer/float conversions on integers, which degrades the performance.

The effectiveness of loop blocking depends on the size of the matrix, as well as the block size. Therefore, you should experiment with different sizes for both of these parameters, to see how they affect the performance. Use the constant `BLOCK` as the block size. This constant is defined in the `Makefile`, just like the matrix size and repeat count are. You can add new parameter configurations by adding new targets in the `Makefile`. For the experiments, it will be convenient to automate the execution of all different variants. You can write a shell script to do this. The output is in CSV format, such that you can process the results using a simple script or in a spreadsheet application.

RQ1. Pick a matrix size `N` so that the three matrices will not fit into the L3 cache (or use cachegrind's `--LL` option to shrink the LL cache size). Would `N=4096` be large enough?

Using cachegrind, investigate which block size results in best cache reuse for this matrix. Demonstrate this using cache statistics.

RQ2. Give the execution time on real hardware for a variety of matrix sizes and block sizes. Include the sizes you investigated in RQ1. Do the results on real hardware agree with the performance simulated by cachegrind?

RQ3. Choose the variant that gave the largest performance improvement in RQ2. Use `perf` to compare real-hardware cache statistics for this variant versus the original program, and show that performance has improved. If your own computer has a very large L3 cache (larger than 8 or 12 MiB), connect to a computer in the lab room instead.

Task 3: Diagonal tile composition

The third problem, `tilecomposite.c`, is another image processing kernel. This program "stamps" an image tile across the diagonal of a background. You may assume that the size of the images are a multiple of 64, but they might not be square. With this simplification, you don't need to check for many corner cases.

Use cachegrind and `cg_annotate` to find places where this program reads the same data multiple times unnecessarily, and make a copy of `op_tile_composite` that **eliminates these redundant reads**. Hint: investigate the macro `image_get_pixel_data`.

RQ1. Show, using profiler output, how you identified redundant reads; and explain how you eliminated them.

Make another copy of the function, and **apply loop blocking** to it to improve the cache re-use of *the tile*.

RQ2. Compare cache statistics (from cachegrind) and the execution time between the blocked and non-blocked version. Discuss the observed results. Restrict your experiment to a background of 4096×4096, and tile sizes of 64 and 512 pixels.

Task 4: Postal code lookup

The fourth problem, `postcode.cpp`, is a database search. Given a list of postal codes (called the *queries*), this program looks up the corresponding street names and cities from a large table. This table takes up around 50 MiB of memory: too big to fit into the L3 cache on most computers. The postal code database, as well as other test data, can be found in the following directory: `/vol/share/groups/liacs/scratch/ca2023/lab3data/postcode/`.

Your task is to investigate possibilities to improve the cache performance of the (large) postcode table. You are not allowed to block, hash, or sort the result table (which contains the queries).

We will pretend that the queries are not known beforehand, so this sort of preprocessing is not possible. All optimization effort must thus be focused on the postcode table.

Currently, `postcode.cpp` contains two search functions: one based on a simple linear scan, and another using a hash table. We will consider various strategies for improving the performance. First, make a copy of the linear scan function and **apply loop blocking** to it.

RQ1. Show that just applying blocking to the linear scan function already gives a measurable performance improvement, and explain why.

RQ2. Think of other ways to improve the blocked linear scan (for example, improving cache reuse or reducing memory accesses) and implement it. The code should still be based on a linear scan. How effective is your strategy?

Instead of using a blocked linear scan, **implement a variant that uses binary search**, with and without loop blocking. A binary search function is already available in `support/postcode.h`.

RQ3. Investigate how binary search performs, and compare the blocked and non-blocked variants. Explain the results.

Finally, **apply loop blocking to the hash-based variant**.

RQ4. Compare the performance of the blocked and non-blocked variants of the hash-based search. Hint: set the number of hash buckets to twice the block size.

Submission and Assessment

Teams may be formed that consist of at most *two* persons The **deadline** is Friday, December 1, 2023. Submit your assignments according to the instructions below.

The maximum grade that can be obtained is 10. The grade is the sum of the scores for the following components, maximum score between square brackets:

- **[2.0 out of 10]** Overall quality of the report.
- **[1.5 out of 10]** Task 1 (grayscale).
- **[2.0 out of 10]** Task 2 (matrix multiplication).
- **[2.0 out of 10]** Task 3 (diagonal tile composition).
- **[2.5 out of 10]** Task 4 (postal code lookup).

The following needs to be submitted:

- The source code of the implemented optimizations. You need to include all different optimized variants. Make sure that the source code that you submit contains all files needed to build the programs.

  Please run `make clean` before making the archive; do not include large files (such as executables or data files). The archive should be smaller than 250 KiB.

- A well-organized report (preferably written in LaTeX) in PDF-format. Some guidelines:
  - Make sure to include your name and student number in the report.
  - Document the hardware (CPU info, cache configuration, etc.) and software (Linux version, compiler version, etc.) that were used for the experiments. Make sure it is clear which cache sizes were used in the experiments, also when cachegrind was used.
  - Present results in a table, graph, or figure; do not put screenshots or copy-paste the terminal output of cachegrind or `perf` into your report. Interpret the results yourself and present these to the reader in a clear manner.
  - Make sure tables and figures have a descriptive caption.
  - If you do include source code, program output or terminal output in the report, make sure to use a monospace font (e.g., `verbatim` environment).
  - Remember to give an explanation for the results, and draw a conclusion from the experiments. Only presenting the results in the report is not sufficient.

Ensure that all files that are submitted (source code, report, etc.) contain your names and student IDs! The source code tarball and PDF-report are to be submitted as *separate* files. Create a "gzipped tar" archive of your source code as follows:

```
tar -czvf assignment3-sXXXXXXX-sYYYYYYY.tar.gz assignment3/
```

Substitute `XXXXXXX` and `YYYYYYY` with your student IDs.
Use the filename `report-sXXXXXXX-sYYYYYYY.pdf` for your report.
Submit the tar-archive and your report through the Brightspace submission site. When submitting the files in Brightspace, write your names and student IDs in the text box. *If you work in a team, ensure only one team member submits the assignment, such that there is a single submission per team!*

The use of text or code generated by ChatGPT or other AI tools is **not allowed**. You are required to implement the requested code transformations **yourself**, and to write your **own** text for the report. All submitted reports and source code will be subject to (automatic) **plagiarism checks** using Turnitin and/or MOSS. Suspicions of fraud and plagiarism will be reported to the Board of Examiners.