Computer Architecture Fall 2022
Assignment 3: Optimizing Single-Thread Performance

Deadline: Wednesday, December 7, 2022 ~~Friday, December 2, 2022~~

In this third assignment we are going to look into optimizing single-thread performance. There are several ways by which execution time can be reduced, among which improving utilization of CPU caches, reducing the number of memory accesses and reducing pipeline stalls by applying instruction scheduling. Note that this corresponds to the material discussed in the textbook in the chapters on Memory Hierarchy Design and the first part of Instruction-Level Parallelism.

In this assignment, we will study optimizations that affect memory performance: improving CPU cache utilization and reducing the number of memory accesses. A problem set consisting of four problems is given. For each problem, you are provided a baseline implementation and a set of research questions. It is your task to answer these research questions. This includes quantifying the performance improvement of possible optimizations by comparing this to the baseline. You will write up your findings in a report. The deliverable will consist of the report and the modified materials that you used to conduct the experiments.

Note that all analyses in your report *must be* backed with measurement results. We take nothing for granted! Just reporting execution time before and after optimization is not sufficient. We also expect you to include measurements of the memory performance in your report to support your argument. Metrics that may be of interest include the number of memory references, cache miss rate, number of instructions executed, etc. These can be obtained using `cachegrind`, see also below. Also the correctness of your optimized code is assessed, when your code is 20 times as fast, it must of course still produce the correct result and not take shortcuts that lead to incorrect results.

Since analysis and optimization is the main aim of this assignment, quality of the delivered code will not be assessed. Truth be told, in the real world most code that is being experimented with is a mess. Only once the solution and effective optimizations have been found, a cleaned up and well-readable code is written.

Background

Within this assignment we will evaluate the effectiveness of caching optimizations on different problems. It is sometimes quite hard to do so on desktop hardware powered by x86 CPUs. This is due to the fact that multiple processes are active at a given time, all making use of the cache, and due to aggressive prefetching performed by the CPU. Therefore, for most problems you will conduct experiments based on simulation (using cachegrind as cache simulator) and experiments on the real hardware are restricted to measuring execution time. For two problems we will ask you to also collect caching statistics on real hardware by sampling hardware performance counters. We expect that you include caching statistics obtained using these methods in your report. A short document on the use of `cachegrind` and hardware performance counters for software performance analysis on real hardware can be found on BrightSpace. Additionally, as an example a brief report on a short investigation of the application of loop interchange on a matrix-vector problem can be found on BrightSpace.

For each problem you are provided a baseline code. These baseline codes encapsulate a particular computation that we would like to study. As computations we will consider matrix multiplication, an image operation and a database problem. Consider that in real-world problems, most of the execution time would be spent in such computations (principle of locality). Such "core" computations are typically referred to as computational kernels. Regularly, you will find that kernels execute in only a few milliseconds. To be able to take good measurements of execution time, the kernel is usually repeated a number of times (say 50 times, or 500 times, depending on the execution time of the particular kernel). The baseline codes provide the means to do so.

You will need to make multiple modifications to these kernels and compare the performance of different modified (optimized) codes to the baseline. We strongly recommend to make a copy

of the baseline code and modify this copy for each problem. After making a copy of the file a new target can be added to the `Makefile`. This way you can create separate executables for the baseline and the variants with optimizations, which will be very helpful when performing the experiments.

The `grayscale` and `tilecomposite` kernels are image operations. A framework is included that can read and write PNG files and that simplifies the manipulation of image pixels. A brief explanation of some aspects of this framework can be found in Appendix A.

Things to keep in mind

Before describing the four problems, we list some important things to keep in mind. A common guideline is that no special compiler flags that control optimization may be used besides `-O3`, `-mtune`, `-mcpu`, `-march`. The `-fno-tree-vectorize` and other `-f` flags present in the Makefile should remain in place; this is to ensure that any speedup you see is not due to the vectorizer suddenly being able to vectorize parts of the code after modification.

Cachegrind

When performing experiments with cachegrind, note the following:

- Disregard the execution time of the program. Cachegrind interprets the program to infer the cache statistics. It does not properly simulate the latencies of the different caches. Therefore, the execution time is *not* a reflection of the actual execution of the program.
- Only repeat the execution once.
- Avoid *huisuil*, because this is a server system with a large cache. This might skew your results/expectations. When working remotely, connect to a computer in the lab room instead.
- Similarly, 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.
- Do not look at the overall statistics reported by cachegrind. These are highly influenced by the surrounding initialization code. Instead, use `cg_annotate` to zoom in to the statistics of the actual compute kernel. (Alternatively, you could set the repeat count to, for example, 500, to have the compute kernel dominate the statistics. However, this will cause the runtime of cachegrind to increase significantly).
- Cachegrind only simulates two cache levels: L1 and LL. By default, it uses the size of the detected L3 cache for the LL cache. If you want to get an idea of the performance of the L2 cache, you can configure LL to act as L2 with the option: `--LL=262144,4,64`.

Using perf

And when collecting cache statistics on the real hardware using perf:

- In particular for the smaller matrices, you need to increase the repeat count to make the program run long enough for perf to collect samples.
- perf runs need to be repeated, results averaged (or median) and standard deviation reported. You will observe that perf never reports exactly the same results. The same holds for measurement of execution time.
- Also consider a matrix size such that the three matrices do not fit L3 cache. Would `N=4096` be large enough?
- To exclude measurement of the initialization code when using perf, use the `-D` option.

- The `-x ,` option to perf will result in output that is easier to post-process with a script or spreadsheet.
- perf is not available on *huisuil*, so when working remotely, connect to a computer in the lab room instead.
- 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.

Problem Set

Problem 1: Gray scale

Investigate the effectiveness of loop interchange for the `grayscale` kernel. Involve *multiple* image sizes in your experiment. You may assume that the input images will always have dimensions that are a multiple of 64, but are not necessarily square. Test data can be found in `/vol/share/groups/liacs/scratch/ca2022/lab3data/`.

The `grayscale` program has a command line option to configure the repeat count and outputs the results in CSV format. With a simple bash script, you can invoke `grayscale` multiple times for different image files and a certain repeat count. Collect all CSV output in a text file for further processing. In your investigation we expect that you prove the effectiveness of loop interchange in three ways:

- RQ1: Show the effectiveness by presenting relevant cache performance statistics from cachegrind.
- RQ2: Compare program execution time.
- RQ3: Show the effectiveness by presenting cache performance on the real hardware as collected using perf.

Also, make sure to explain the results! Why is there an improvement or reduction in performance? Do these match your expectations?

Problem 2: Matrix Multiplication

Take the basic implementation of matrix multiplication (`matrixmul`) and apply loop blocking. Refer to the lecture slides for an example of loop blocking. On BrightSpace, a more in depth explanation of loop blocking is available as well. **DO NOT USE `fmin()`, as this will negatively affect the performance!**.

The effectiveness of loop blocking depends on the size of the matrices and the block sizes. Consequently, both of these should be parameters for your experiment. This means that you want to perform experiments using *multiple* matrix sizes and *multiple* block sizes. Make a copy of `matrixmul.c` before implementing loop blocking. Make the block size a parameter, using a define similar to the matrix size `N`. In the `Makefile`, you can create multiple targets to generate multiple executables that use different parameters. To conduct a series of experiments, you can simply execute these binaries one after the other (consider writing a bash script). Capture the output in a text file. The output is in CSV format, such that you can write a script to process the results, or use a spreadsheet application.

- RQ1: Pick a matrix size (`N`) such that the problem will not fit your L3 cache (or reconfigure the LL cache used by cachegrind using `--LL`). Investigate which block size results in best cache reuse using cachegrind. Demonstrate this in your report with memory performance statistics obtained using cachegrind.
- RQ2: For different matrix sizes and block sizes, give the execution time on real hardware. Include the matrix and block sizes used in your answer to RQ1. Do the improvements in execution time concur with the improvements in (simulated) memory performance?

- RQ3: Pick the variant for which the largest performance improvement was found in RQ2. Compare cache statistics obtained with perf for this variant and the baseline code, to show that the cache performance has indeed improved on the real hardware.

Problem 3: Diagonal Tile Composition

The `tilecomposite` program implements a kernel that performs tiled image compositing over a diagonal. You may assume that the input image will always have dimensions that are a multiple of 64 (but are not necessarily square). This property will greatly simplify optimization, since many corner cases do not have to be checked and do not have to be dealt with. We want to optimize the memory performance of this program. Investigate the following two research questions:

- RQ1: can you discern from the profiler output (cachegrind, `cg_annotate`) whether there are redundant reads of datastructures? Answer this question using profiler output. Secondly, eliminate these redundant reads. Hint: consider using the `register` C-keyword and `image_get_pixel_data`.
- RQ2: optimize the function for cache re-use of *the tile* using loop blocking. Report on the cache statistics (cachegrind) and execution time before and after optimization. Discuss the observed results. Restrict your experiment to a background of 4096 by 4096 pixels and tile sizes of 64 and 512 (suitable test data can be found in `/vol/share/groups/liacs/scratch/ca2022/lab3data/`).

Problem 4: Finding street names for postal codes

We would like to find the streetname and city for a sequence of postal codes (postcodes). Provided is a big table (rather array) containing all postal codes, streetnames, city names and provinces. This table, sized approx. 50 MiB when loaded in memory, does not fit L3 cache on laptops and desktops. We will refer to the sequence of postal codes to resolve as the query. Test data for this problem can be found in `/vol/share/groups/liacs/scratch/ca2022/lab3data/postcode/`.

The problem of mapping postal codes to street and city names can be seen as a join-query using the postal code as key. Your task is to investigate possibilities to improve the memory performance. This includes reducing the number of memory access and improving cache reuse. For this particular problem, this is a delicate trade-off between LL cache misses, number of memory references, number of executed instructions and algorithmic complexity. Consider these in your analysis. All of these also affect execution time.

You are giving the program `postcode` which implements support code to load the table and query in memory and help with benchmarking. Also given are a basic (and very naive!) linear scan variant and a variant using a hash table. For the optimizations, there is the restriction that the result table (which is initially filled with the query), may not be blocked, hashed, or sorted. Only sequential scans of the result table are allowed. We focus here on optimization of the memory performance of the (large) postcode table.

- RQ1: Given the basic and very naive linear scan, show that simple loop blocking already gives a measurable performance improvement. Explain why.
- RQ2: What other optimizations can you think of to further improve the blocked basic scan code? Consider improving cache reuse and reducing memory accesses. Your code should still be based on scans.
- RQ3: Implement a variant using binary search and a variant using binary search and loop blocking. A binary search function for you to use is present in the `support/postcode.h` header file. Analyze whether loop blocking improves performance for binary search. Explain your results.

- RQ4: Given the variant using a hash table, construct a loop blocked variant of this. Analyze its performance and explain the results obtained. (Hint: set the number of hash buckets to twice the block size.)
- Finally, summarize your thoughts. Were there any notable results? Was there anything that deviated from your expectations? Which code was the fastest; and what would be the reason why?

Submission and grading

Teams may be formed that consist of at most *two* persons. The following needs to be submitted:

- Source code of the implemented optimizations. Note again: we want to see the code for each task individually. And please ensure to not include binaries, object files or image files in your submissions to keep the size of the file archives manageable. Use `make clean`! If the size of the tarball containing your source code is larger than 250Kb, you probably did something wrong.
- A well-organized report (preferably written in LaTeX) in PDF-format. Some guidelines:
  - Ensure to report on what hardware (CPU type, cache configuration) and software (Linux distribution name and revision, compiler version) the experiments were carried out!
  - Present results in a table or figure, not by pasting screenshots in your report.
  - Do not paste verbatim output of cachegrind or perf into the 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).
  - Conclusions must be drawn from experiments. Just 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. Please create a "gzipped tar" archive of your source code:

```
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. Also note your names and student IDs in the text box in the submission website. *Please, ensure only one team member submits the assignment, such that there is a single submission per team!*

Deadline: Wednesday, December 7, 2022 ~~Friday, December 2, 2022~~

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]** Problem 1.
- **[2.0 out of 10]** Problem 2.
- **[2.0 out of 10]** Problem 3.
- **[2.5 out of 10]** Problem 4.

All submitted reports and source code will be subjected to (automatic) plagiarism checks.

A Some details on the image operation kernels

The two-dimensional images are stored in a one-dimensional array. To find the pixel in memory for a location (x, y) the formula $A[x + W * y]$ (column-major) or $A[y + H * x]$ (row-major) is used. In the file `image.h` the macro `image_get_pixel` is defined which will return a pixel to the requested pixel. The macro `image_get_pixel_data` accomplishes the same, but allows you to specify the data pointer and rowstride directly.

Image arrays contain elements of the type `uint32_t` and each element contains a single pixel. So, each pixel is stored in 32 bits. Within this assignment we are using the `RGBA8888` pixel format. This means that each pixel consists of four channels that take eight bits each: red, green, blue and alpha. Red is stored in the most significant byte and alpha in the least significant byte. In order to extract the value of the red channel, the following code can be used:

```
uint32_t pixel = *image_get_pixel(image, x, y);
int red = ( pixel >> 24 ) & 0xff;
```

The alpha channel indicates the opacity and must remain `0xff` in most cases. Most image operations are performed in the floating-point domain. However, our image arrays consist of integer elements. To solve this, pixels are unpacked and packed when required. Packing is the encoding of a vector $\langle r, g, b, a \rangle$ as a single integer. Unpacking is the inverse operation. The reason why the operations are performed in the floating-point domain is that in the floating-point domain all channels have a 32-bits precision, while this is only 8-bits within the packed integer domain. Within `image.h` a number of useful macros is defined that simplify these operations. The following example illustrates these macros:

```
/* Retrieve a pixel from A, multiply this with a constant and write back
 * to the image array.
 */
uint32_t pixel = *image_get_pixel(image, x, y );
rgba_t pixel_rgba, factor, result;

RGBA(factor, .5f, .5f, .5f, 1.f);
RGBA_unpack(pixel_rgba, pixel);
RGBA_mult(result, pixel_rgba, factor); // result = pixel_rgba * factor
RGBA_pack(pixel, pixel_rgba);

*image_get_pixel(image, x, y) = pixel;
```

The type `rgba_t` is used to represent a vector consisting of four float components. These components can be accessed individually using the member fields `x, y, z, w`, which correspond in that order with the channels r, g, b, and a. So, `my_rgba.z` gives the blue channel value.