Computer Systems
Fall 2018
Optimization Lab: Performance Improvement
Assigned: Oct. 10, Due: Fri. Oct. 19, 20:00

1 Introduction

In 1989, Donald Foster attributed "A Funeral Elegy for Master William Peter" to William Shakespeare based on a stylometric computer analysis. The attribution received much attention and was accepted into the canon by several highly respected Shakespeare editors. However, analyses published in 2002 by Gilles Monsarrat and Brian Vickers showed that the elegy more likely was one of John Ford's non-dramatic works, not Shakespeare's, a view to which Foster conceded. In this lab, we focus on repeating these experiments, aiming to find an algorithmic way to compute this association. For this, we use text classification.

Text classification is the process of grouping text documents into one or more predefined categories based on their content. A common approach for text classification is to use the k-Nearest Neighbor classifier algorithm. This classifier works by computing the similarity between a document to classify and a dataset of already classified documents. The document is then classified looking at the labels of the k most similar documents.

Our classification uses word frequency as features. We have a labeled dataset, where we have the word frequencies of several manuscripts by several authors, and we will measure the similarity of the unlabeled manuscript (in this case, "A Funeral Elegy for Master William Peter") against these word frequencies. The unlabeled manuscript will be attributed to the author of the most similar labeled manuscript.

This procedure can be quite expensive in terms of performance, especially when there are many features (i.e., we look at the frequencies of many words) and many manuscripts (i.e., we need to compute similarity against each manuscript).

One of the key elements here is the use of the right similarity metric. We propose here, as example, three such similarity metrics, with increasing complexity: Manhattan Distance (MD), Euclidean Distance (ED) and Cosine Similarity (CS).

The goal of this lab is to improve the performance of this classification (i.e., text classification based on the k-Nearest Neighbors algorithm (kNN)) as much as possible. Three optimization stages must be attempted, and they could eventually be combined for even better results:
a. Sequential code optimization.

b. SIMD code parallelization.

c. Multi-threading.

You will have to submit your best effort for each stage, but we strongly recommend you try to combine all these optimizations, preferably during stage (c). Our estimate is that the combined effect of all the optimizations you have applied should give at least 5x improvement (i.e., a speedup of at least 5) over the original version when measured on the reference machine (i.e., a node in DAS5) using 4 threads.

2 Logistics

As you already know, the performance of an application depends not only on the algorithm and the (size of) input data, but also on the machine itself. To make sure that everyone uses the same hardware, we provide everybody with accounts on the DAS5 cluster (http://cs.vu.nl/das5). The measurements for the grading are to be done on DAS5. Instructions on how to use DAS5 are included separately.

However, it is strongly recommended you develop your code and expand your experiments to your local machine, too.

You have access to three DAS5 clusters:
fs1.das5.liacs.nl
fs3.das5.tudelft.nl
fs0.das5.cs.vu.nl
We recommend you use the first two as much as possible, and resort to using the third one only if absolutely necessary.

NOTE: the three sites share the same user credentials - i.e., you can login with the same name and password - but have separate file systems. Thus, you have to copy the files to every site you intend to use.
NOTE: It is not recommended to keep switching sites when measuring performance.

3 Downloading the assignment

The initial code for the optimization lab is available for download on Canvas.

To access DAS5, you *must* be part of the UvA network - by either being at UvA or using the UvA VPN.

To work on the assignment, our recommendation is to develop the solution on your own machine, and, once you are satisfied with its performance, copy it on the DAS5 for performance measurement. We strongly recommend you include both measurements (local and DAS5) in your report. While only the DAS5 numbers are absolutely necessary for your final grade, including a more detailed performance analysis increases your score.

You can also work directly on DAS5, with three main challenges: (1) you have one account for the group, not individual, (2) you will need to use vim, nano, or emacs as an editor, and (3) you will have to be connected via VPN/UVA network at all times. To login to the DAS5 and work remotely:

```
ssh <das5account>@fs1.das5.liacs.nl
```

If you work locally, in order to perform the final measurements, you will need to upload your files to the DAS5. To get your files onto the DAS5, you should use scp (in a Terminal window). For example:

```
scp -r <../optimizationlab> <das5account>@fs1.das4.liacs.nl:
```

NOTE: this will overwrite the files on the DAS5 *without warning*. Make sure you backup your code locally and/or on DAS.

NOTE: you can use any of the three DAS5 sites mentioned earlier, not only fs1.

4 Files and rules

The optimizationlab folder should now contain multiple files: some input data (.csv files), templates for the code you need to write, Makefiles, and a script for benchmarking. A more detailed description of the files follows.

Do not modify the **k_nearest.c** file and the **Makefile**, they provide a reference implementation and compiling flags.

You can modify the **Makefile.students.mk**, if you want/need to, but all modifications have to be clearly explained in the Makefile.students.mk file itself (as comments) and in the report. **Any performance improvement due to Makefile.students.mk changes must also be explained clearly and related to specific optimizations.**

You will be modifying the following files:

k_nearest_seq.c , k_nearest_simd.c, and **k_nearest_thread.c .**

To compile the code (all three files) use:

```
linux> make clean
linux> make
```

You might encounter the following problems:

- the gcc compiler is not defined on your machine. Check which gcc compiler you have and use that (typical for Macbook's).
- some of the flags in the Makefile are not working for your machine.
- some of the flags in the Makefile are too "conservative" for your machine - this is especially true for vectorization. Please check what your CPU supports - by checking the /proc/cpuinfo file. Look for "avx2"; else for "avx"; else for "sse4". Use -mavx2, -mavx, or -msse4 respectively in the Makefile.

To execute the files, you will run the benchmarking script: driver.py. This will launch each version of the application multiple times (default: 10) and provide performance data to be used for grading. Specifically, we report the performance obtained by the reference implementation and the performance the optimized version has obtained. Finally, the benchmarking script reports *speed-up*, measured as the ratio between the execution time of the reference implementation and the new, optimized one.

This speed-up is the measure of your success: the higher, the better!

5 The assignment requirements

We have used 34 books (some of the entries are repeated) from several authors (Shakespeare and Ford included) to create the **dataset*.csv** file. Each row of this file represents a book, each column contains a metric (i.e., term frequency–inverse document frequency) that is correlated to the frequency of a word in the book. We have included 3 datasets: dataset_3, _4_ext, and _5_ext_repeated. We measure performance for the largest one (dataset_5_ext_repeated.csv), but you can use the smaller ones for debugging purposes.

The kNN algorithm uses three different ways to compute the distance between books: Manhattan Distance (MD), Euclidean Distance (ED) and Cosine Similarity (CS).

A sequential implementation to calculate those distances is given in the **k_nearest.c** source file, and repeated (as reference), in the other files. In general, MD is given as an example. **You do not have to provide ANY changes or optimizations for the MD calculation.**

This lab has three phases. You are required to:

- **Phase 1**: Optimize the sequential implementation **of the ED and CS** metrics, using the techniques from Chapter 5. (**3 pts**)
- **Phase 2:** Use Single Instruction Multiple Data (SIMD) to **optimize the computation of the ED and CS metrics** and, optionally, of the classification itself. The MD SIMD implementation (for both AVX (simd_avx) and SSE (simd)) is given to you as an example and does not need changing. (**4 pts**)
- **Phase 3:** Use multi-threading to optimize the classification. An example of the Multi-Threading implementation is provided for the MD metric (**3 pts).**

We award points for each phase based on **correctness** (1,1,1 points for Phases 1,2,3 respectively) and **speedup** compared with the reference implementation.
A speed-up per phase of 1.5x, 2x, 2x will guarantee 7 points (2,3,2 points per Phases 1,2,3, respectively).
Additional points are given for exceeding these expectations. For example: achieving higher speed-up, combining the optimizations, in-depth performance analysis.
The final grade is an average of the grades you achieve for ED and CS.

6 Lab Book

You will report on your lab approach, challenges, and lessons learned into a lab report.

For this assignment, you will receive a template of such a lab report, and you will be requested to answer around 10 questions in the provided text (a bit like ”fill in the blanks”, only with longer answers). You will be graded for these answers only, but make sure they are coherent, they make sense in the context of the report you submit, and they are readable. We do not correct grammar mistakes, but text that is not readable will not be graded.
Each correct answer should not be longer than the specified number of words, but can include images or graphics or code (not counted as words) if needed. Once you have filled in your answers, generate the PDF file of the report using your favorite Latex compiler (e.g., online, Overleaf or SharedLatex). The PDF of your report is the file you hand in as your lab report.

Like most of the course material, the lab report is provided in English. If you have problems writing your answers in English please inform your TA or contact the coordinators (Giulio, Ana) asap.

7 Grading

You can use the provided driver.py for the grading. Note that there is a flag (ON_DAS5) that needs to be set to "False" or "True", for local benchmarking or running on DAS5, respectively. Moreover, the number of runs (REPEAT_MEASURE_NB) over which the average is done can be reduced for faster experiments.

We grade both metrics (ED and CS) separately. The grade you obtain is the average of the two grades. Per phase, speed-ups of around 1.5x, 2-3x (depending on SSE or AVX), and 2-3x (depending on the hardware), respectively, are to be expected. Again, correctness and speed-ups of at least 1.5x, 2x, 2x will guarantee a grade of 7/10 for the code for each metric.

Note that, for grading, the measurements for **k_nearest_thread** are performed using 4 threads only.

The code must compile and run correctly!

As always, the code and the report are graded separately.

8 Hand-In

You will hand-in the code and the report separately.
For the code, please hand-in one archive on Canvas, containing:

- k_nearest.c -- unmodified
- Makefile.students.mk - optionally modified
- k_nearest_seq.c , k_nearest_simd.c, k_nearest_book.c, and k_nearest_threads.c -- the files (partially) modified by you.
- any other file you modified.

For the report, please upload the PDF directly.

9 Notes and recommendations

Phase 1:

- For Phase1, consider the typical optimizations such as code-invariant loop motion, avoiding code-blockers, reduction in strength, loop unrolling, accumulators, etc.
- Feel free to apply algorithmic optimizations, too - i.e., replacing more complex computation with lower complexity alternatives with the same functionality.
- You can experiment with compiler flags, but note that you still need to explain the optimizations that have been applied and bring evidence (e.g., assembly code snippets) that the optimizations have been indeed performed by the compiler.

Phase 2:

- There are two options to implement SIMD optimizations: using the book method or using intrinsics. You are free to choose whichever you want: the intrinsics version will provide better performance, but requires lower level programming. Examples of both are provided for the MD distance: in k_nearest_simd.c for the intrinsics version, and k_nearest_simd_book.c for the book version. Please clearly state in your report which of the two options you are using.
- In case you use the k_nearest_simd_book, depending on the architecture, you might observe a slow-down. If the implementation is correct and you document the reasons for this slow-down, you will still get the full points.
- Note that driver.py uses, by default, the intrinsics version. You can either rename the executable or change driver.py to use the book version.
- You may choose to implement both versions and compare the results. This will most likely give you additional code points.

Phase 3:

- You can adjust the number of threads and/or study its correlation with the number of threads your machine has.
- You can attempt an alternative implementation, using OpenMP, to compare against your pthread-based multi-threaded solution. This will again give you additional points.

In general:

- You can obtain performance some improvement by smartly using compiler flags (thus, modifying the `Makefile.students.mk`). If you choose this path, note that it is expected you *explain* and bring proof (usually, assembly code and its analysis) that the optimization you performed was actually done well.
- Phases need not be tackled in order.
- Make sure you record all the optimization **ideas** you have, even if you did not manage to implement them all. They will be useful when writing your report. For example: if you think there is a better data structure that would improve the performance of the code, but will disrupt too much of the code, you can make an argument for using it, explaining why, how, and what would be the expected impact.
- Go as far as you can into performance analysis, e.g., for reasoning, with proof, why a certain optimization has not met the expected results. For example, if you are observing that your code does not perform well when using 8 threads, and you think this can be because you only have 2 cores, discuss this and bring proof that the machine you are running only has 2 cores.
- We strongly recommend combing several/all optimizations to achieve the highest performance.
- Bonus points are awarded if you find and fix bugs in the code. The bugs need to be documented in the code and in the report.