Computer Systems
baiCOSY06, Fall 2016
Optimization Lab:
Understanding Performance Optimizations
Assigned: Oct. 12, Due: Fri. Oct. 21, 11:59

Ana Lucia Varbanescu(a.l.varbanescu@uva.nl) is the lead person for this assignment. Giulio Stramondo, Janosh Haber, Kyrian Maat, Tycho van der Ouderaa, Joop Pascha and Thomas Schaper are the teaching assistants.

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 three examples, with increasing complexity.

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 approaches can be used, and they should be combined:

a. Sequential code optimization.

b. SIMD code parallelization.

c. Multi-threading.

Submit your best effort for each stage. We are looking for the best performing code. However, the combined effect of all the optimizations you have applied should give at least **3x** improvement over the original version.

2 Downloading the assignment

You can find the file optimizationlab-handout.tar at the location /opt/prac/cs_ai/optimizationlab on the machine acheron.fnwi.uva.nl.

The same archive can also be downloaded from the blackboard page of the course.

Start by copying optimizationlab-handout.tar to a protected Linux directory in which you plan to do your work. Then give the command

```
linux> tar xvf optimizationlab-handout.tar
```

This will create a directory called optimizationlab-handout that contains a number of files.

Do not modify the **k_nearest.c** file, this one provides a reference implementation.

You can modify the Makefile, if you really want/need to, but all modifications have to be clearly explained in the Makefile file *and* in the report.

You will be modifying the following files, in this order:

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

To compile the code (all three files), you will 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).

- the acheron compiler does not recognize the -mavx2 option. In this case, make sure you use -msse3. To find out what the CPU supports, check the /proc/cpuinfo file. Look for "avx2"; else for "avx"; else for "sse4". Use -m*** in the Makefile.

- if you cannot run on the virtual machine or on your own system or on acheron, please email Ana (a.l.varbanescu@uva.nl) for an account on different machine.

To execute the files, you will run:

k_nearest_seq, k_nearest_simd, and k_nearest_threads

Each file will report the performance obtained by the reference implementation and the performance your optimized version has obtained. Finally, it will report 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!

3 Description

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 on the largest one (dataset_5_ext_repeated.csv).

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 of those distances is given in the **k_nearest.c** source file, and repeated (as reference), in the other files.

This lab is made of 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 is given to you as an example. (**8 pts).**
- **Phase 3:** Use multi-threading to optimize the classification. An example of the Multi-Threading implementation is provided (**4 pts).**

We award points for each phase based on **correctness** (1,2,1 pts for Phases 1,2,3 respectively) and **speedup** compared with the original given sequential implementation. An overall speed-up of 3x will guarantee 10/15 points. Additional points are given for example, for any of the following: higher speed-up, additional optimizations, in-depth performance analysis, etc.

4 Lab Book

The lab book should **clearly** reflect the design and implementation work for each phase; each optimization should be traceable and reproducible, and its impact clearly marked. The report must contain **why** and **how** the optimizations where done.

The problem definition is supposed to address also this question:
Why are the performances different for each of the given metrics?

The lab book will be evaluated on content, structure, wording and completeness, as described in

http://www.practicumav.nl/onderzoeken/labboek.html

The course staff will inspect your code.

The maximum grade for this lab is 30 points.

The maximum grade for the lab book is 15 pts. However, additional *design points* and *analysis points* can be awarded (in the limit of the overall 30 points) to compensate a code that scores below par. We do this to

incentivize you to think of code optimization both at design and implementation levels.

5 Hand-In

When you have completed the lab, you will hand in one archive on Blackboard, containing *at least*:

- k_nearest.c -- unmodified
- Makefile - optionally modified
- k_nearest_seq.c , k_nearest_simd.c, and k_nearest_threads.c -- modified by you.
- labbook.pdf -- your labbook.

If easier, you can also archive the full folder of your implementation. Make sure, in any case, that you clearly mark your names and student numbers in the code, and your family names in the folder name.

6 Notes and recommendations

- For Phase1, consider the typical optimizations such as code-invariant loop motion, avoiding code-blockers, reduction in strength, loop unrolling, accumulators, etc.
- For Phase3, you can adjust the number of threads and/or study its correlation with the number of threads your machine has.
- Design points are awarded for interesting application design ideas which could not have been implemented (due to time or complexity). 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.
- Analysis points are awarded for performance analysis - that is, 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.
- 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.
- When you formulate research question(s), do not forget that your goal is not only to apply some textbook techniques, but to select the ones that are suitable and understand their impact.

- Make sure that, for any optimization you apply, you explain why and how you applied it. Also important, make sure you explain, based on the empirical evidence, whether the impact was as expected and, if not, why not.
- Do not discard optimizations that do not behave as you expect. They can still give you analysis points.
- Do not discard optimizations that you think are hard to implement. They can still give you design points.