Computer Systems
baiCOSY06, Fall 2017
Lab Assignment : Optimizing Text Similarity Analysis
Assigned: October 12, Due: October 19, 20:00.

Nik (11058269), Tim (11848782)

1 Introduction

The goal of this lab is threefold. First, the exercises prove how different ways to improve the performance work in practice. Second, to learn about the most common forms of concurency found in most machines today: SIMD and multi-threading. Third, to learn about performance variabiltity in practice, trying to see how different elements (compilers, flags, the way we right code, etc) impact performance.

The lab is setup as follows: three code versions are provided, and the all need to be improved such that the overall performance of the application is maximized. The three versions are originally independent, but they can also build on top of each other. Each code focuses on a specific set of performance optimization techniques: optimizing sequential code, using SIMD to increase the efficiency in exploring ILP and the SIMD units in the CPU, and, finally, using multiple threads to make efficient, explicit use of the multiple cores existing in the processor.

This report is a description of the techniques and results obtained in each phase. Thus, for each phase, we report: ideas at design level (i.e., what are possible optimizations), implementation details (i.e., how were they implemented), a brief effectiveness analysis (i.e., analysing whether they work or not, by assessing how much improvement they achieved, and why), and ideas for future work.

2 Performance optimization for the sequential code

2.1 Background

Q1. Briefly explain known techniques to improve the performance of the sequential code.

1. Loop unrolling:
   Doing multiple steps per iteration makes it so that you have to loop less time, speeding up the program.

2. Reducing function calls:
   By replacing functions by code you allow the compiler to see what is going to happen and apply

optimization on the code. What a function does can not immediately be seen by the compiler so the compiler can't optimise it.

3. Loop-invariant code motion:
   By moving some operations outside of the for loop and storing them in variables you can avoid doing the same operation every iteration when this is not necessary.

2.2 Design: useful optmizations for k_nearest

Q2. Describe the optimizations you thought would have worked for k_nearest, at conceptual level, and why they should have worked. Very brief. We thought that loop unrolling and replacing function would improve the speed of the k_nearest program because these techniques are often not used because it makes for difficult to read code. Next to that making use of SIMD would improve the speed because many operations are done. Finally using multiple threads would also improve the speed because it would make it possible to do multiple parts of the program at the same time.

2.3 Implementation and results

Q3. Describe (in detail, with pseudocode when needed) how you have implemented the different optimizations you have tried, one by one. We started implementing the sequential optimization methods because these were the easiest to understand starting by unrolling all loops where this is possible. After that we replacing functions by operators (for example mul(x,y) by x*y). For the other methods we looked at the examples given with the manhattan distance and tried to apply this to the eucledian and cosine. Next to that we applies the sequential methods to some parts of the SIMD and Thread files to further improve the performance. Q4. Describe and analyze how the performance was impacted by each optimization. Please analyze all results, positive, neutral, or negative. That is: not only list the numbers, but attempt to reason whether they meet your expectations and, if not, why.

1. Sequential: Speedup ED: 1.02 Speedup CS: 1.30
   This shows that these methods indeed has some effect on the performance but it is not a huge improvement. This is to be expected as we don't use any better hardware.

2. SIMD: Speedup ED: 2.54 Speedup CS: 4.48
   This shows how powerful the new hardware specialized for operations on whole vectors is. It's a big improvement, as to be expected.

3. Multi Thread: Speedup ED: 2.55 Speedup CS: 3.58
   Here we use multiple threads which also leads to a significant improvement in performance as multiple parts of the program can now be done at the same time. The improvement is not as much as the specialized hardware as with SIMD, but still a big improvement.

3 Using SIMD

3.1 Background

Q5. Briefly explain what SIMD is and how it can be used to improve performance for sequential code. SIMD is a method which can be used when doing the same operation on multiple data points. SIMD can do this operation on a whole vector of data points at the same time. You can use this by vectorizing your data and then letting the special hardware do the calculations instead of normal ALUs.

3.2 Design: SIMD for k_nearest

Q6. Describe the approach to SIMD for k_nearest, at conceptual level, and why it should work. Specific SIMD challenges should be discussed. Just like with loop unrolling you search for operations which are not dependant on each other, this is the case in multiple for loops in the k_nearest code. By putting the data points used during those operations in the vector you can do said operation on the whole vector at the same time saving time, while afterward you jump further with each iteration depending on how many datapoints could be handled in one vector operation.

3.3 Implementation and results

Q7. Describe (in detail, with pseudocode when needed) how you have implemented SIMD, one by one. By looking at the example given for the manhatten distance we managed to apply the SIMD method to eucledean and cosine. Just like with the loop unrolling we check how many data points can be used during one iteration and tweak the for loop so that it iterates using the right steps while using SIMD to do the operations on the data points. Q8. Describe and analyze how the performance was impacted by SIMD. Please analyze all results, positive, neutral, or negative. That is: not only list the numbers, but attempt to reason whether they meet your expectations and, if not, why. SIMD: Speedup ED: 2.54 Speedup CS: 4.48 The performance improved by a lot because SIMD uses specialized hardware to do a lot of operations at the same time so there was a big improvement in the runtime.

4 Using Multiple Threads

4.1 Background

Q9. Briefly explain what multi-threading is and how it can be used to improve application performance. Multi threading is where you use multiple threads on the same program to be able to execute multiple parts of the same program in parallel.

4.2 Design: multi-threading for k_nearest

Q10. Describe the approach to multithreading for k_nearest, at conceptual level, and why it should work. Specific multithreading challenges should be discussed. Multi threading was possible here because a lot of

operations are done that are not dependent on each other so by splitting the load onto multiple threads these operations can be done in parallel.

4.3 Implementation and results

Q11. Describe (in detail, with pseudocode when needed) how you have implemented multi-threading, one by one. We implemented multi threading by looking at the example given for the Manhatten distance and applying the same method to cosine and eucledean. Q12. Describe and analyze how the performance was impacted by multi-threading. Please analyze all results, positive, neutral, or negative. That is: not only list the numbers, but attempt to reason whether they meet your expectations and, if not, why. Multi Thread: Speedup ED: 2.55 Speedup CS: 3.58
The performance improved quite a bit, for ED similar to using SIMD and for CS much better than sequential but a little worse than SIMD. Parallel processing is very strong but specialized hardware (SIMD) is even better.

5 Conclusion and Future Work

5.1 Final results

Q13. Briefly discuss your final, overall results, and provide a brief list of your interesting findings about code optimization. From the results we can conclude that multiple optimization methods are useful but to get significant improvement you need to make changes to the hardware. An interesting finding is that for ED SIMD and multi threading showed the same performance boost and we are still not quite sure why this is while for CS SIMD was better. Sequential optimization is especially useful because it can be applied to almost every situation.

Future work

Q14: What are the next steps to improve the performance of the code even further? Or to improve the analysis? Next steps to be taken are to see how multiple optimization methods could be used in combination with each other to see what combination gives the best performance improvement. Also we could look at different type of problems where maybe one optimization method is viable while others wouldnt work.