ADVANCED SYSTEMS LABORATORY
BAUM-WELCH ALGORITHM PROJECT REPORT

Josua Cantieni, Franz Knobel, Cheuk Yu Chan, Ramon Witschi

Department of Computer Science, ETH Zürich, Switzerland

ABSTRACT

Machine learning has become increasingly popular in both academia and industry, with numerous real-world applications of steadily growing importance. With that comes the need for very fast implementations of ubiquitous models, done right. In this paper, we propose a case study of the well-established Baum-Welch algorithm, reaching a much higher performance than possible using just compiler optimizations, given a baseline implementation. Thus, we show that sophisticated system's engineering is crucial to help researchers and practitioners save time, and resources, to focus on the things that matter most.

1. INTRODUCTION

In computer science and statistical calculation models, the Baum-Welch algorithm [1] (an expectation-maximizing algorithm) is used to find the unknown parameters of a Hidden Markov Model (HMM). The maximum likelihood estimates and the posterior marginals for the given parameters (transition and emission probability) of an HMM are calculated with the emissions as inputs.

The motivation to do a case study of the Baum-Welch algorithm came from its applications in various fields such as bioinformatics [2] [3], robotics [4], and statistical computing. It also finds its application in the field of speech recognition [5] [6] [7].

The structure of the algorithm entails large parts of independent computations. Thus vectorization and other optimizations become feasible. Our main contribution, therefore, is to show that we can gain significant speed improvements while providing a detailed empirical analysis of how the algorithm scales for various input dimensions.

Considering related work, the problem has long been studied. Hence, implementations in various flavors and programming languages already exist. "Some Mathematics for HMM" from Dawei Shen [8] contains a similar, but more prototypical implementation in Matlab, which also has the hmmtrain[1] function. For Python, there are two libraries, hidden_markov[2], and hmmlearn[3]. For the former, the computations are done via matrices to improve the algorithm runtime. In Julia, the package HMMBase[4] supports arbitrary univariate and multivariate distributions. Widely known for its use for statistical computing and graphics, R has an HMMFit[5] function, using the gradient descent algorithm to minimize the negative log-likelihood function. But the most holistic implementation we found was the GHMM library [9] in C.

The first step was to understand the algorithm and how it worked (section 2). After the textbook implementation functioning as a baseline, we implemented various optimization methods and checked for correctness (section 3). Lastly, based on empirical benchmarks, we performed a sophisticated analysis (section 4).

2. BACKGROUND

To understand how the Baum-Welch algorithm can be optimized, we discuss first how the algorithm works and give a detailed description of the cost analysis of the algorithm. Code and calculus examples of the following subsections are deliberately omitted as they can be read in detail in the linked sources.

The Baum-Welch algorithm. is a special case of expectation maximization. Through iteration, hidden variables in a given hidden Markov model (HMM) are learned with the forward-backward algorithm [10].

2.1. Hidden Markov Models

The HMM consists of multiple smaller models called Markov chain models (MCM). They display probabilities of sequences of random variables, called states. Each of these states can take on values from some set containing words, tags, or even symbols representing anything (e.g., weather). The MCM

[1] https://www.mathworks.com/help/stats/hmmtrain.html

[2] https://pypi.org/project/hidden_markov/

[3] https://github.com/hmmlearn/hmmlearn

[4] https://github.com/maxmouchet/HMMBase.jl

[5] https://www.rdocumentation.org/packages/mhsmm/versions/0.4.16/topics/hmmfit

makes the strong assumption that to predict the future in the sequence, all that matters is the current state. The states before have only an indirect impact on the future via the current state. It's as if to predict tomorrow's forecast one could examine today's state without being allowed to look at yesterday's [11].

2.2. Expectation-Maximization

Expectation-Maximization, also called the EM-Algorithm, is an approach for performing maximum likelihood estimation in the presence of latent (hidden) variables. It does this by first estimating the values for the latent variables, then optimizing the model, and lastly, repeating these two steps until convergence. It is an effective and general approach and is most commonly used for density estimation with missing data [12].

2.3. Forward-Backward Algorithm

It is a dynamic programming algorithm for learning/updating latent variables [10]. It consists of the following steps, with $0 < t \leq T$ and T as the number of time steps.

1. **The forward step.** computes the probability for the system being in a hidden state at time t, given all observations until time t.
2. **The backward step.** computes the probability of all further observations (t+1 until T), happening under the assumption the system is in a specific hidden state at time t.
3. **The combination step.** multiplies the outcomes of the forward and backward step together to calculate the probability, for the system being in a hidden state at time t, given all observations.

These steps will be reflected in the Baum-Welch algorithm.

2.4. Cost Analysis

The cost analysis is in flops per cycle and was calculated under the assumption of an Intel Haswell processor. Hence, the processor has the following ports for floating-point operations [13]:

1. FMA, MUL, DIV
2. FMA, MUL, ADD

All operations counted in the baseline algorithm resulted in the following:

$9TKN^2 - 5KN^2 + N^2 + 8TKN + 3KN + K + 2KNM + 2TK + N + NM$

With $\#add > \#mul > \#div$
`K` = number of observation sequences / training datasets
`N` = number of hidden state variables
`M` = number of distinct observations
`T` = number of time steps

2.5. Verification

We checked the correctness of the baseline implementation via explicit test cases, which were cross-validated with the ghmm library [9]. Then, we follow the paradigm of checking each subsequent optimization performed back to the baseline. Additionally, we have general tests that randomly initialize K, N, M, T, observations, as well as initialization, transition, and emission probabilities. We check after convergence, whether the probabilities sum to one, indicating conceptual correctness. Crucially it can be shown that the general expectation-maximization algorithm always has a monotonic decrease of the negative log-likelihood after each iteration of the algorithm [14]. We verify that this is the case, for each possible test case. Lastly, given the baseline implementation and the calculated probabilities and auxiliary arrays, e.g., for scaling, we cross-check whether these arrays contain the same numbers up to an epsilon for each optimized implementation. With all of the mentioned procedures, we are confident about the correctness of all implementations performed.

3. METHODS

Many diverse optimization steps were carried out to achieve a quick implementation of the Baum-Welch algorithm. Throughout the process, the assumption was made that N, M, K, and T are divisible by 16, and additionally, should T be at least 32 for more efficient and better optimizable implementations.

3.1. Textbook implementation

Our first attempt at the baseline was the standard theoretical implementation of Wikipedia and other textbooks. However, this had the problem of numerical instability since the respective probabilities were passed over time. These will leads to the collapsing of the probabilities, resulting in undesired events such as NaN's or wrong outcomes. Therefore, this implementation was discarded.

3.2. Numerically stable implementation

To obtain a numerically stable implementation, the implementation described in [8] and [6] was adapted, in which a method is used where the probabilities are scaled. This implementation has been sufficiently verified to convince ourselves that it produced the right results. This acts as our

baseline for the performance analysis and also as verification for the optimizations that follow.

3.3. Reordering

One of the first observations we noticed is that data dependency exists (e.g., backward step can only be carried out after forwards step). As soon as these were calculated, the temporary variables sigma and gamma could be updated. This allows us to combine various loops. The consequence of this is that loops are used more efficiently, reducing some calls, and also reducing some operations that were initially performed in multiple loops. The order of the nested loops was also largely rescheduled such that, if possible, the memory access should be iterated sequentially in row-major order. That promotes spatial locality.

3.4. Scalar replacement, Precomputing

Another optimization we did was scalar replacement. We have stored elements of the memory that we will use several times in temporary variables and thus performed computations to not only reduce the number of memory accesses but also to enable register assignment and instruction scheduling by the compiler. Code motion and strength reduction were also used, especially for index calculations, but also for some floating-point operations. This only led to a slight improvement in performance, since the compiler also did this internally for the simple cases, and the small improvement was mostly explained by the floating-point operations.

3.5. Blocking

Further improvements toward locality and memory access could be achieved by blocking. These were applied to almost all loops. The greatest impact on performance was found in heavily nested loops such as the forward and backward pass. Due to dependencies, blocking could not be applied to all dimensions for certain functions (e.g., in forward pass). However, empirical observations have shown that blocking across more than two dimensions does not bring about any visible improvements, and in some cases even worsens it. We assume that by blocking the higher layers, more explicit memory accesses were required, and thus certain optimizations on the compiler side were prevented. With this prior knowledge, blocking was also applied to the update functions. Through examinations, we found that the block size of the innermost loop can be increased as desired without sacrificing any noticeable performance. But for the greatest effect, the block size should be divisible by four. Thus, we decided on a block size of 16 based on our initial assumption. Regarding vector optimizations, the block size of the second innermost loop was set to four.

3.6. Unrolling

Unrolling is similar to blocking but easier to implement, as you only work on one loop instead of blocking a loop through implementing another around it. One goal was to prepare for vectorization with unrolling by a multiple of four and to cut repeated loads or calculations with scalar replacement to a possible minimum. Another one was to decrease the number of branch predictions. Furthermore, through unrolling, the scheduler got a larger window of instructions from which to select. This increased instruction-level parallelism. There were some difficulties in terms of dependencies. The unrolling of the number of time steps (T) was not possible due to dependencies to the predecessor (t-1) and the successor (t+1).

3.7. Vectorization

With vectorization, four double floating-point operations can be calculated simultaneously and thus increasing the performance of the algorithm [flops/cycle]. This leads to an increase in spatial locality as four double floating points can now be loaded and stored in one operation because of the cache mechanics. As not every calculation was high in spatial locality, the obstacle was overcome with the combination of unrolling and horizontal operations. Horizontal addition and multiplication were done with the lowest possible number of operations we could think of.

3.8. Alternative Input

At some point, we realized, that the access of an input matrix was always (except one time) column-wise. Through a cost and optimization analysis, we decided to provide the matrix already transposed to the algorithm. Now every access (except a negligible one) was row-wise and thus was easily unrolled for vectorization which increased spatial locality.

3.9. Combined Optimization

Here the goal was to combine all optimizations mentioned before and thus get the highest performance.

4. EXPERIMENTS

In our experiments, we wanted to find out how our optimizations perform. As our algorithm takes four input values, which define the size of the used matrices, the first step was to look at every parameter by itself. Then we shift our focus to the performance impact of different compiler flags and finally to different compilers.

The experiments were performed on a Laptop running an *Intel Haswell i7-4710MQ* with a clock speed of 2.5GHz.

This processor has an L1-data-cache size of 32KB (8-way associative) and an L3-cache size of 6 MB (12-way associative). The system had 12 GB of RAM and was running OpenSUSE Leap 15.0. The code has been compiled in a Docker container featuring a Debian-image but the binaries were run natively on the system.

We used two compilers for our experiments: GCC 9.2.0 and clang 7.0.1.

For both compilers the following compiler flags were used:

- -O3 -ffast-math -mavx2 -mfma -march=native
- -O3 -fno-tree-vectorize -ffast-math -mavx2 -mfma -march=native
- -O3 -funroll-loops -ffast-math -mavx2 -mfma -march=native

The first flags use standard optimizations from GCC/-clang, the second disables vectorization and the third flags allow GCC/clang to further unroll loops on their own to gain more possibilities for optimizations.

In the following sections, we analyze the performance of our optimizations in different situations. To measure the performance we executed all implementations multiple times on the same input data. As the algorithm can converge in different numbers of iterations we always let it run 500 iterations. The final performance is the average for one iteration.

4.1. Comparing the input values

Every input value impacts the performance in two different ways. On one hand, they change the flop count and on the other, they change the size of the input matrices (see section 2.4 about cost analysis).

To measure the impact of one parameter we held the others at fixed values. For `K`, `N`, `M` we chose 16 and for `T` 32 as fixed values.

Figure 1 shows the performance of our optimizations with only increasing `K`. The combination of all optimizations performs the best, followed by our vectorization implementation. Both our blocking and scalar replacement implementation perform better for small `K` than for big `K`. After `K=196` the performance is stabilising.

Next, we have Figure 2 that shows the performance of our optimizations for different `N`. As seen in the cost analysis in section 2.4, `N` influences the flop count and memory in $O(N^2)$. This is reflected in the plot as the performance drastically drops at `N=48` for our best implementation, wherein Figure 1, the performance drop was at `K=128`. At this point, the data does no longer fit into the L3-cache. Another interesting point is that at `N=128` the performance of our vectorized implementation is worse than the performance of the baseline. We were unable to identify the cause of why this happens.

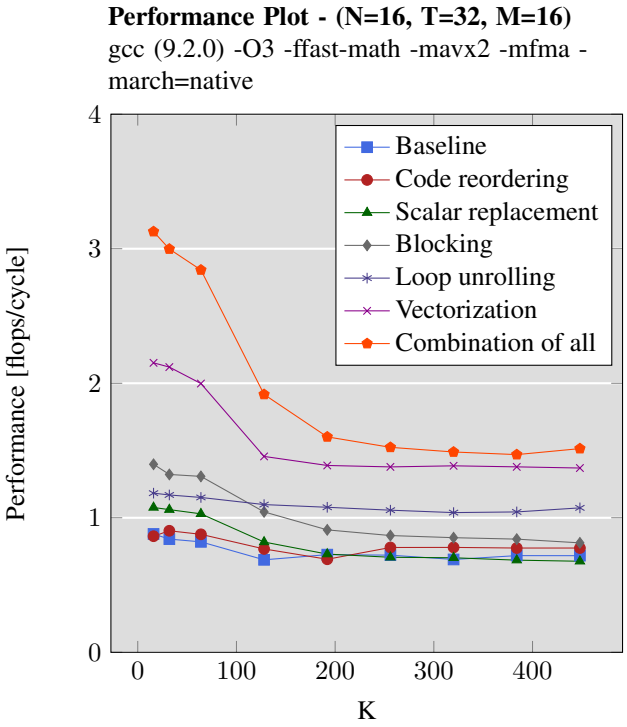

Fig. 1: *Performance of our implementations of the Baum-Welch algorithm with fixed parameters `N,M,T` and variable parameter `K`.*

In section 2.4, it is trivial to see that `M` has very little influence on the overall performance compared to the other parameters. This is why we chose to scale the size of `M` exponential and thus chose a logarithmic scale for Figure 4. The performance decreases steadily and settles at 2^{13}. It is interesting to see that for large `M` our loop unrolling implementation slightly surpasses the performance of our combined implementation. We assume this is due to some optimizations that GCC was able to do to outperform our combined optimization for large `M`.

The performance impact of `T` is not as big as for `K` or `N`, as seen in Figure 3. The performance of the single optimizations behave similar as in Figure 1 for `K`. The most notable difference is that the performance of each optimization is slightly lower for large `T` than for large `K`.

We saw in this section how the different parameters impact the performance. We found out that `N` has the biggest impact on the performance as expected according to our cost analysis of `N`. From our optimizations, the best improvements in performance are vectorization and loop unrolling for big values and blocking for small values. However, the combination of the optimization surpasses every single optimization (except for very large `M`).

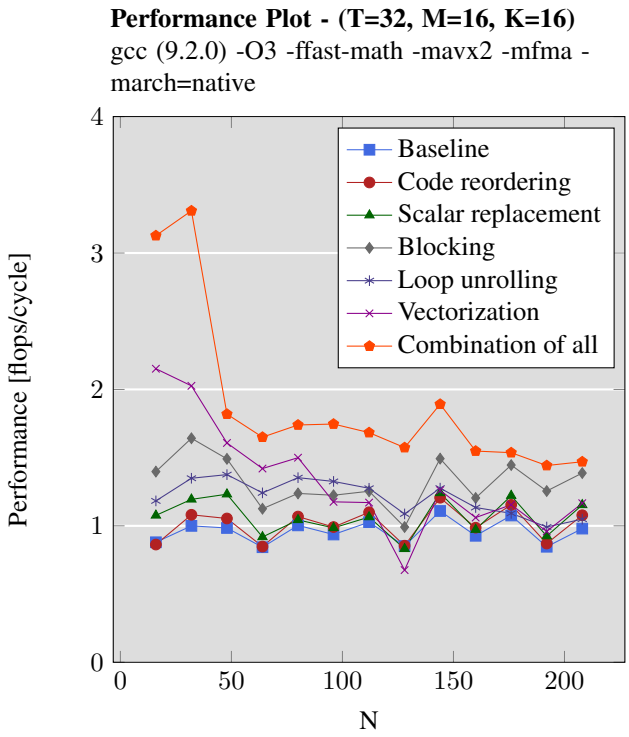

Fig. 2: *Performance with fixed parameters* `K,M,T` *and variable parameter* `N`.

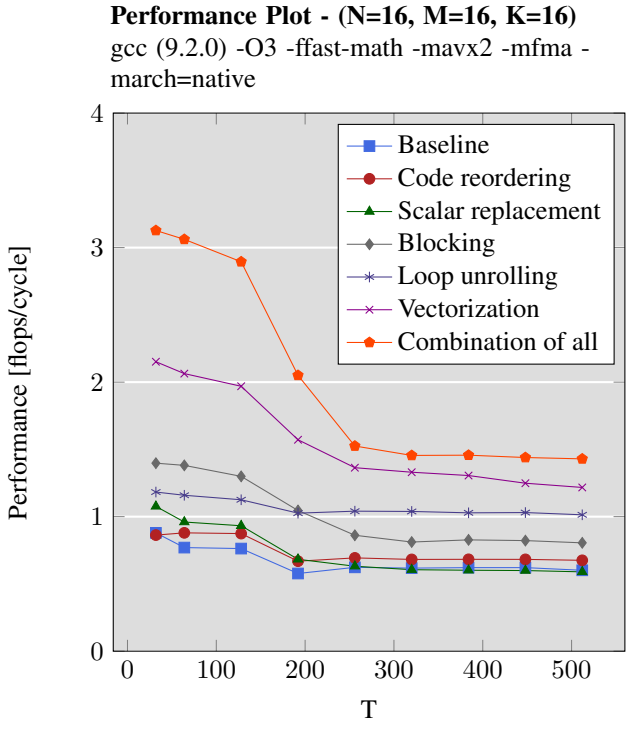

Fig. 3: *Performance with fixed parameters* `K,N,M` *and variable parameter* `T`.

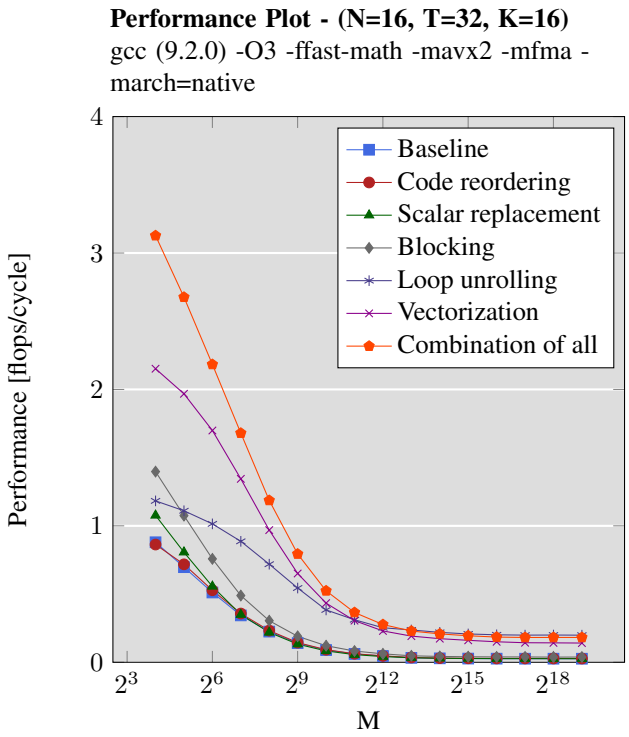

Fig. 4: *Performance with fixed parameters* `K,N,T` *and variable parameter* `M`.

4.2. Comparing compiler flags

Next, we compare the different compiler flags with the focus on the plots of `N`. The plots for the other parameters behave similarly and are not as interesting.

Figure 5 shows our optimizations when we compile them with the compiler flag `-fno-tree-vectorize`, which disables vectorization for GCC. Some interesting things happened here compared to Figure 2. Our blocking and loop unrolling implementation get a performance boost. We don't know exactly why those optimizations perform better with this flag, but we assume that this is due to some overhead that GCC introduced with vectorization. On the other hand, the performance of the baseline is worse than without the flag. We think that GCC could vectorize certain things that improved the performance.

The flag `-funroll-loops` allows GCC to heuristically decide which loops to unroll. Figure 6 shows the impact of using this flag. It is notable that compared to Figure 2, the performance of all implementations is shifted up slightly. For example, the performance of the baseline never drops below one in this case. The algorithm uses many loops, so GCC has found some loops that could be further unrolled to gain performance in all implementations.

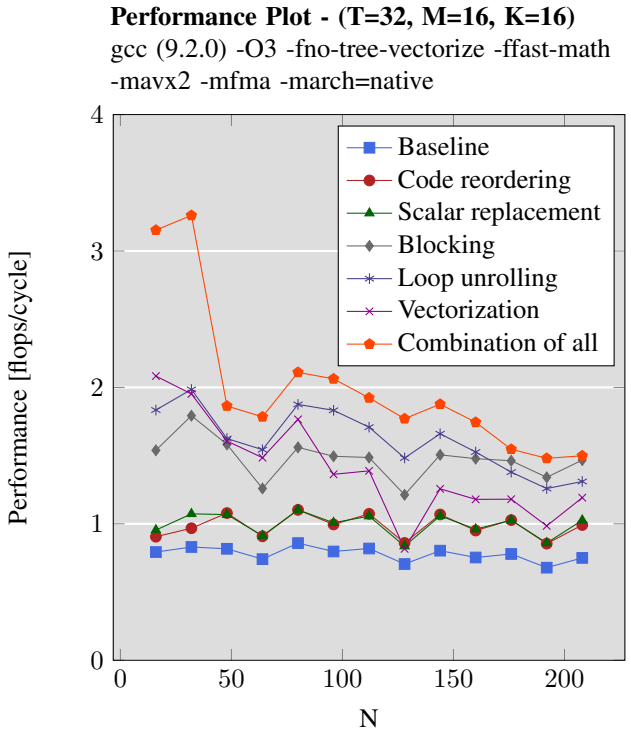

Fig. 5: *Performance with fixed parameters `K,M,T` and variable parameter `N`.*

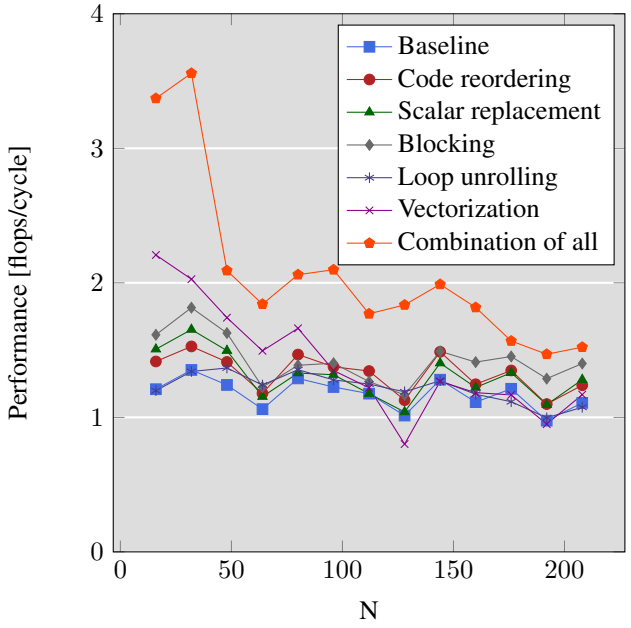

Fig. 6: *Performance with fixed parameters `K,M,T` and variable parameter `N`.*

4.3. Comparing compiler

For now, we only looked at how GCC performs. We wanted to know how our code performs if we use a different compiler, so we compiled everything with clang 7.0.1. Again we only use `N` to compare against GCC as the other parameters behave similarly.

Figure 7 shows the performance of our implementations when compiled with clang. Compared to Figure 2, which shows the performance for GCC, it is notable that the performance of our loop unrolling implementation is almost as good as our combined optimization. We think this is due to clang vectorizing our loops better than GCC. Also, the performance of our combined, our vectorized, and our blocking implementation are better than with GCC. On the other hand, the performance of the baseline, the reordering, and the scalar replacement implementation are worse than with GCC. This makes us believe that clang is better with optimized code than GCC but not as good with unoptimized code.

Of course, we also tried the compiler flags on clang. Figure 8 shows the performance when compiling with clang using the `-fno-tree-vectorize` flag. Most notable is that our loop unrolling implementation is still almost as good as our combined optimization, which is strange because it is not vectorized, and the peak performance of the processor on which the benchmark has been run is two flops per cycle for non-vectorized implementations. Our loop unrolling implementation surpasses three flops per cycle. We found out that this is due to clang still vectorizing certain instructions when possible. This is called superword-level parallelism (SLP) and can be disabled using a special flag. Due to time constraints, we did not try the benchmarks with this flag to test the impact.

At last, we also tested the flag `-funroll-loops` with clang. Unlike with GCC, we did not notice any major differences, as seen in Figure 9, when using this flag with clang. It seems that clang is already doing a good job at unrolling loops by default or that it does not have enough data to do more with this flag.

Interesting to see in all plots for `N` with both GCC and clang is that our vectorization implementation has that drop at `N=128` for which the cause is unknown.

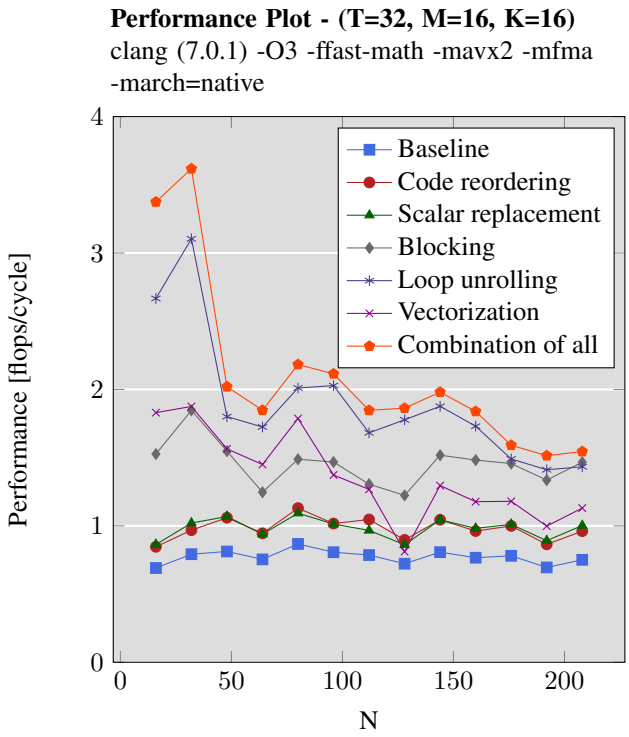

Fig. 7: *Performance with fixed parameters* K,M,T *and variable parameter* N.

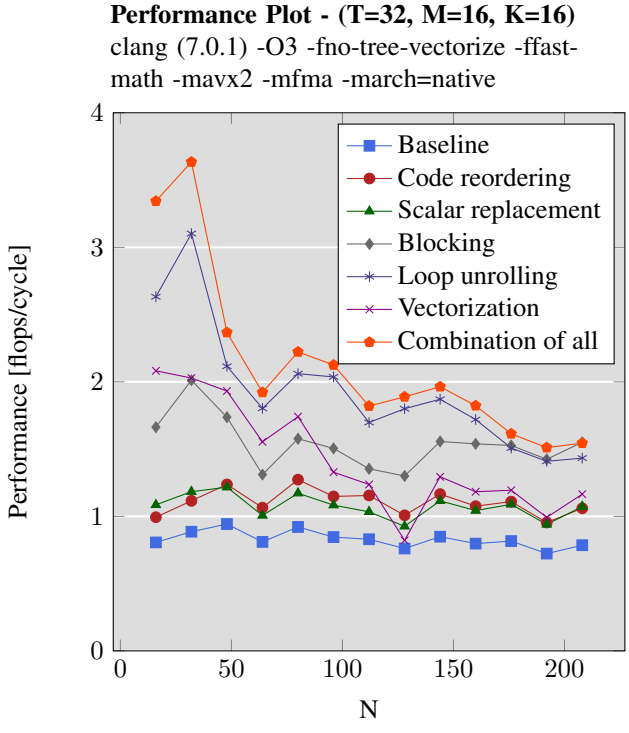

Fig. 8: *Performance with fixed parameters* K,M,T *and variable parameter* N.

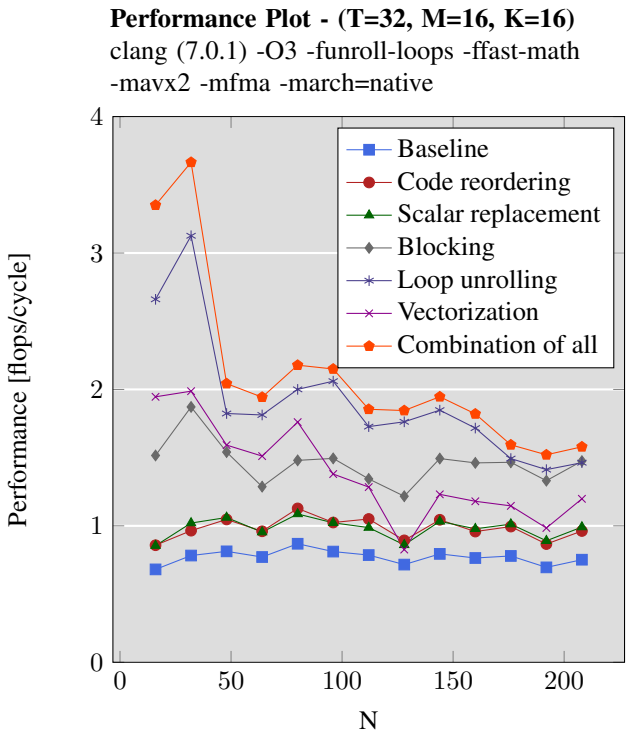

Fig. 9: *Performance with fixed parameters* K,M,T *and variable parameter* N.

At last, we did a roofline plot for both GCC and clang to see how they compare. For this, we used the performance for $K = 16$, $N = 16$, $M = 16$, $T = 32$, and the standard flags. Figure 10 shows the roof-line plot for GCC and Figure 11 shows the roof-line plot for clang. In those figures, we see that our scalar implementations are compute-bound and that our vectorized implementations are on the edge between being memory-bound and compute-bound for the smallest size we ran our implementations on. Those plots also show well how clang is better with optimized code than GCC and how GCC is better with unoptimized code than clang.

In the roofline plots, we marked the performance, which we calculated to be the peak performance for scalar and vectorized implementations. We calculated the number of cycles that are needed if we could distribute all instructions to all CPU-ports such that they are always busy. We reached approximately two-third of each peak performance. We think that there is not much we can do to improve the performance due to data dependency between the different steps. Notable is also that our scalar implementation of loop unrolling surpasses the performance peek for scalar implementations in the clang plot. As already mentioned, this is due to clang performing superword-level parallelism, which transforms instructions into vectors, so it is indeed vectorized then.

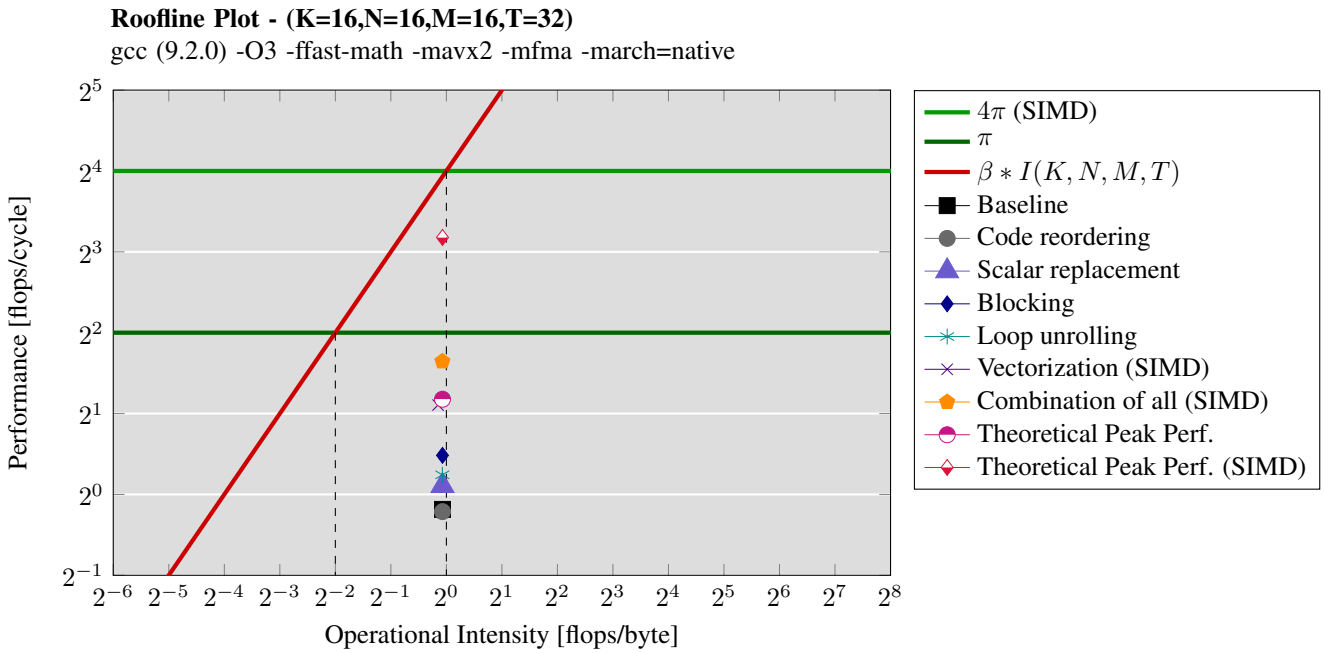

Fig. 10: *Roofline plot for GCC. β indicates the memory bandwidth, which is 32 bytes per cycle for our processor, and π the peak performance for scalar or vectorized implementations.*

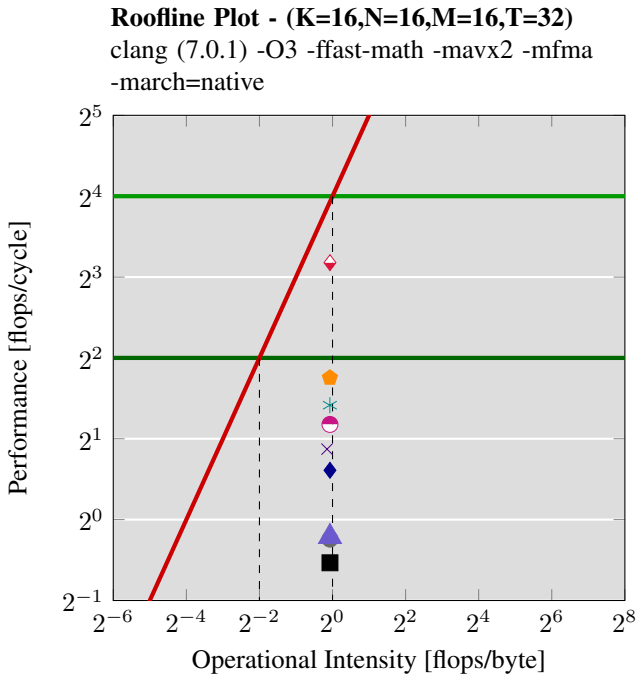

Fig. 11: *Roofline plot for clang. The legend is the same as in figure 10.*

5. CONCLUSIONS

In this paper, we made a case study of the Baum-Welch algorithm, and empirically analyzed our findings with various benchmarks. We found that the combinations of optimization techniques make a difference in the potential gains achievable. Furthermore, the choice of compiler and compiler flags thereof can change, or even worsen the performance. Thus, it is important to carefully engineer a good selection of parameters to reach the full potential of the algorithm on given computer architecture.

6. CONTRIBUTIONS OF TEAM MEMBERS

In the following, our contributions to the optimization and their analysis of our project are listed.

Cheuk Yu Chan. He was responsible for the implementation of the scalar optimized algorithm, in which various methods were combined and applied. He provided the interface for the benchmark, which was then expanded further in the process. Together with Ramon Witschi, the baseline implementation was created, which served as the basis for the analysis and further optimizations.

Franz Knobel. Worked close with Josua Cantieni (online code-sharing[6]). Focused on combining all optimizations and if it was necessary redid some (scalar optimization, reordering, loop unrolling, vectorization) from the start

[6] https://www.codepile.net/

so that they all fit together. Created a script to easily generate clear and concise graphs (roofline, performance) from the experiment raw data.

Josua Cantieni. He did one version of the reordering of the algorithm without scalar replacement. Moreover, he worked with Franz Knobel on the loop unrolling version and combining the different optimizations. Together with Franz Knobel, he created the scripts that generate the Graphs from the data gained from the benchmarks. He was responsible for the benchmarks and ran all benchmarks on his laptop.

Ramon Witschi. His main contribution can be found in the verification process of the algorithm to make sure it is correct given the assumptions made. Furthermore, he provided a fully vectorized implementation and contributed various bug fixes and changes in other parts of the code as well.

7. REFERENCES

[1] Biing-Hwang Juang Lawrence R. Rabiner, "An introduction to hidden markov models," 1986.

[2] S. Kasif O. White S. L. Salzberg, A. L. Delcher, "Microbial gene identification using interpolated markov models," https://www.ncbi.nlm.nih.gov/pmc/articles/PMC147303/pdf/260544.pdf, 1998.

[3] E. C. Powers S. L. Salzberg A. L. Delcher, K. A. Bratke, "Identifying bacterial genes and endosymbiont dna with glimmer," https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2387122/pdf/nihms-48173.pdf, 2007.

[4] Wang Cong et al., "Learning mobile manipulation through deep reinforcement learning," *Sensors (Basel)*, vol. 20, no. 3, 2 2020.

[5] R. Mercer F. Jelinek, L. Bahl, "Design of a linguistic statistical decoder for the recognition of continuous speech," https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1055384, 1975.

[6] L.R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," https://www.ece.ucsb.edu/Faculty/Rabiner/ece259/Reprints/tutorial%20on%20hmm%20and%20applications.pdf, 1990.

[7] T. Masuko T. Kobayashi T. Kitamura K. Tokuda, T. Yoshimura, "Speech parameter generation algorithms for hmm-based speech synthesis," https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=861820, 2000.

[8] Dawei Shen, "Some mathematics for hmm," https://courses.media.mit.edu/2010fall/mas622j/ProblemSets/ps4/tutorial.pdf.

[9] "Ghmm library," http://ghmm.sourceforge.net/index.html.

[10] Michael Collins, "The forward-backward algorithm," http://www.cs.columbia.edu/~mcollins/fb.pdf, Last accessed: 01.06.2020.

[11] Daniel Jurafsky & James H. Martin, "Speech and language processing," http://web.archive.org/web/20200222061910/https://web.stanford.edu/~jurafsky/slp3/A.pdf, Last accessed: 01.06.2020.

[12] Jason Brownlee, "A gentle introduction to expectation-maximization (em algorithm)," https://machinelearningmastery.com/expectation-maximization-em-algorithm/, Last accessed: 01.06.2020.

[13] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[14] Sean Borman, "The expectation maximization algorithm - a short tutorial," https://www.cs.cmu.edu/~dgovinda/pdf/recog/EM_algorithm-1.pdf.