BAUM-WELCH OPTIMIZATIONS

*Luca Blum, Jannik Gut, Jan Schilliger**

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

In this paper, the Baum-Welch algorithm for hidden Markov models gets analyzed, so that a numerically stable and fast implementation can be presented. First, some numerical issues get highlighted and resolved, afterwards a theoretical estimation on the runtime gets presented and some designs of reordering the steps are shown. Those designs and other, more general strategies then got implemented, tested for the impact of every parameter and presented in the form of another theoretical count, performance plots, time plots, a roofline plot and a Cachegrind analysis. During this analysis it is shown, that the optimized working set is no more dependent on the amount of observations, the amount of instructions could be lowered to a twentieth and a peak speedup of 10x performance was achieved.

1. INTRODUCTION

A part of the course "Advanced Systems Lab" is to implement the methods taught to write fast code, measure and reason about it in a project. Our group chose the Baum-Welch algorithm[1], since all of us are interested in the domain of machine learning, where this algorithm is used to optimize hidden Markov models[2], which are as an example used to predict sequencing errors of DNA.[3]
After the introduction, the algorithm will be presented (section 2) and tricks to stabilise it are shown, before some optimization steps are presented (section 3), which then get tested in the last chapter (section 4) before coming to a conclusion (section 5).
The Baum-Welch algorithm is an algorithm, which is classically bound by memory, especially if, like in this case, double precision floating points are used to calculate probabilities. There are advanced algorithms[4], which have optimized the math part to only use memory in O(*N*), instead of the classical O(N^2), where *N* is the amount of hidden states, but since this paper here focuses on optimizing code and not mathematics, the straight forward algorithm was used as base for a new implementation, which is stable, thanks to another paper,[2] and easy to use for parameters, which are divisible by four, so it can be used as a benchmark and starting point for further research.

*We would like to thank Markus Püschel for the lecture "Advanced Systems Lab" and the team behind the lab, especially our project mentors Eric P. Stavarache and Gleb Makarchuk.

2. ALGORITHM BACKGROUND

2.1. Algorithm structure

Variables. The Baum-Welch algorithm needs three main parameters: the amount of hidden states (*N*), the different observables that can be emitted (*K*) and the amount of those signals captured (*T*). The algorithm is an estimation maximization algorithm, that tries to optimize the transition matrix (a), which has size ($N \times N$) and gives the probability to jump from some current state to some other in the next time step, the emission matrix (b) of size ($K \times N$) gives the probability of the emission of some observable from some hidden state and the initial state probability(π) with size (*N*), which gives the probability of some state at the start of the signal capture according to the *T* observations (*Y*).
Algorithm procedure. The Baum-Welch algorithm consists of many iterations. Each iteration takes as inputs the transition matrix, emission matrix and initial state probability, then goes through three main steps: the forward step, the backward step and the update step, until it can output updated versions of the inputs. If the probability that those matrices generate the sequence of observations increases by a large enough amount, the next iteration starts with the output of the previous iteration until convergence.
Forward step. The forward step starts at time $t = 1$ and goes until $t = T$ at each time step generating *N* different α.

$$\alpha_i(t) = \Pr[Y_1 = y_1, \ldots, Y_t = y_t, X_t = i|\theta]$$

$$\alpha_i(1) = \pi_i b_i(y_1)$$

$$\alpha_i(t+1) = b_i(y_{t+1}) \sum_{j=1}^{N} \alpha_j(t) a_{ji}$$

Backward step. The backward step goes the reverse direction; it commences at time $t = T$ creating *N* distinct β each time and finishes at $t = 0$.

$$\beta_i(t) = \Pr[Y_{t+1} = y_{t+1}, \ldots, Y_T = y_T|X_t = i, \theta]$$

$$\beta_i(T) = 1$$

$$\beta_i(t) = \sum_{j=1}^{N} \beta_j(t+1) a_{ij} b_j(y_{t+1})$$

Update step. With the help of the intermediate variables γ and ξ the new transition matrix (a^*), new emission matrix (b^*) and new initial distribution (π^*) get computed.

$$\gamma_i(t) = \Pr[X_t = i|Y, \theta] = \frac{\alpha_i(t)\beta_i(t)}{\Pr[Y|\theta]}$$

$$\xi_{ij}(t) = \Pr[X_t = i, X_{t+1} = j|Y, \theta]$$

$$\xi_{ij}(t) = \frac{\alpha_i(t) a_{ij} \beta_j(t+1) b_j(y_{t+1})}{\Pr[Y|\theta]}$$

$$\pi_i^* = \gamma_i(1)$$

$$a_{ij}^* = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

$$b_i^*(v_k) = \frac{\sum_{t=1}^{T} 1_{y_t = v_k} \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)}$$

2.2. Stability improvements

Since there are potentially many hidden states and time steps, the corresponding probabilities might be small enough to run into problems with numerical stability, that is why some stability improvements[2] have been implemented.

Scaling trick. Looking at α, one can see that the sum of probabilities decreases at each timestep, even though summing the first sum in $\alpha(t)$ over all states gives the same probability as from $\alpha(t-1)$; the multiplication by b diminishes the probability.

$$\alpha_i(t+1) = b_i(y_{t+1}) \sum_{j=1}^{N} \alpha_j(t) a_{ji}$$

If *T* is big, this causes issues with numerical stability, that is why to counter this effect, a scaling factor c_t is introduced, with which every $\alpha(t)$ gets scaled by, to ensure that the sum of all $\alpha(t)$ stays constant.

$$c_t = \frac{1}{\Sigma_{i=1}^{N} \alpha_i(t)}$$

$$\alpha_i(t) = c_t \alpha_i(t)$$

$$\Sigma_{i=1}^{N} \alpha_i(t) = 1$$

To make sure that the results do not get skewed, $\beta_i(t)$ also has to get scaled by the same c_t.

Sum of log probabilities. The overall probability, $(\Pr[Y|\theta_n])$, that the current versions of the matrices fit the observations is usually computed:

$$\Pr[Y|\theta_n] = \Sigma_{j=1}^{N} \alpha_j(t)\beta_j(t)$$

But since the probabilities can get very low and the exact value might be important to decide whether to stop the loop or do another iteration, a more stable version, to take the log of probabilities, is used:

$$log(\Pr[Y|\theta_n]) = -\Sigma_{t=1}^{T} log(c_t)$$

As a result the desired precision of the algorithm also has to get scaled.

Timing stability. To ensure that timing results are stable and not greatly skewed by some (un)lucky choice of starting parameters, the maximal amount of iteration steps has been fixed and kept track of to divide the overall running time by it to get the time of a single iteration, which then, depending on the need for precision can be multiplied by the estimated amount of steps.

2.3. Theoretical measurements

The analysis assumes $K \approx N < T$ and dominated terms are omitted.

There are $3KNT + 10N^2T$ flops comprised of $3TN^2 + KNT$ adds, $6TN^2$ mults and $TN^2 + KNT$ divs and KNT comparisons. A different implementation[4] has time in O(TN(K+N)) per iteration, which can be correlated to the amount of flops.

The standard are $2KNT + 14N^2T$ memory accesses which are comprised of $11TN^2 + 2KNT$ reads and $3TN^2$ writes. Every read and write to an array has been counted, so the numbers are under the assumption of no cache to keep it simple and invariant of system and parameters.

The working set is the last metric provided which is $TN + 4T + N + K$. This is the spot where the different algorithm shines, as it managed to get a working set in O(N).[4]

3. OPTIMIZATIONS

There are three big leaps forward documented in building the fast implementation (C-optimizations, reordering and vectorization), also instead of vectorization there is a version that uses the BLAS [5] library instead of manually unrolling and vectorizing that shows some difficulties of the Baum-Welch algorithm.

3.1. C-optimizations

As a first step, simple C-optimizations were applied. This includes scalar replacement, inlining and log weakening.

Scalar replacement was particularly useful by saving recurring divisions as a constant. This allowed to multiply with the constant instead of using a division. Further log_{10} was used to compute the log-likelihood for the finishing criteria in the first version. Since the exact value of the log-likelihood is not needed, only it's change, log_{10} could be

exchanged with the faster log_2. Additionally, some experiments with pragma ivdep were made to eliminate memory aliasing, which turned out to be no improvement. Even the most standard implementation only used pointers to arrays, so no abstract data structures could be replaced in the project.

3.2. Reordering

omit $\frac{1}{\Pr[Y|\theta]}$. In most cases ξ and γ are used in a division with each other and are both divided by the same constant factor $\Pr[Y|\theta]$.

$$a_{ij}^* = \frac{\Sigma_{t=1}^{T-1}\xi_{ij}(t)}{\Sigma_{t=1}^{T-1}\gamma(t)} = \frac{\Sigma_{t=1}^{T-1}\frac{\alpha_i(t)a_{ij}\beta_j(t+1)b_j(y_{t+1})}{\Pr[Y|\theta]}}{\Sigma_{t=1}^{T-1}\frac{\alpha_i(t)\beta_i(t)}{\Pr[Y|\theta]}}$$

$$b_i^*(v_k) = \frac{\sum_{t=1}^{T}1_{y_t=v_k}\gamma_i(t)}{\sum_{t=1}^{T}\gamma_i(t)} = \frac{\Sigma_{t=1}^{T}\frac{1_{y_t=v_k}\alpha_i(t)\beta_i(t)}{\Pr[Y|\theta]}}{\Sigma_{t=1}^{T}\frac{\alpha_i(t)\beta_i(t)}{\Pr[Y|\theta]}}$$

Instead of usually dividing by $\Pr[Y|\theta]$ and cancelling it in the division of sums. Flops can be saved by only dividing where the result does not get in a second division, e.g. for π.

$$a_{ij}^* = \frac{\Sigma_{t=1}^{T-1}\xi_{ij}(t)}{\Sigma_{t=1}^{T-1}\gamma(t)} = \frac{\Sigma_{t=1}^{T-1}\alpha_i(t)a_{ij}\beta_j(t+1)b_j(y_{t+1})}{\Sigma_{t=1}^{T-1}\alpha_i(t)\beta_i(t)}$$

$$b_i^*(v_k) = \frac{\sum_{t=1}^{T}1_{y_t=v_k}\gamma_i(t)}{\sum_{t=1}^{T}\gamma_i(t)} = \frac{\Sigma_{t=1}^{T}1_{y_t=v_k}\alpha_i(t)\beta_i(t)}{\Sigma_{t=1}^{T}\alpha_i(t)\beta_i(t)}$$

Storing sums. Rather than storing each $\xi_{ij}(t)$ and $\gamma_i(t)$ first and then in a second step adding them up to sums over time, it is better to save cache space and I/O costs by adding $\xi_{ij}(t)$ directly to Ξ_{ij} and $\gamma_i(t)$ in Γ_i. There are still a few γ that have to be stored extra, but those are a small minority.

Storing results in *a*,b*.* In the previous version, first γ and ξ get computed and then those results are combined and stored in the same matrices *a,b* that the iteration started with. This second step of copying can be avoided by storing Γ and Ξ already in the correct architecture for *a,b (a*,b*)*. If need be, *a,b* have to be scaled accordingly and the next iteration can start with *a*,b** as *a,b*.

Out of order scaling. To save loads, it is best if values are only loaded if they are immediately needed and if possible execute many operations on them. Usually, the transition matrix gets scaled at the end of an iteration with the overall probability, which can be computed once the whole matrix is already completed. Instead of scaling at the end, it is more efficient to scale at the first time the transition matrix gets used; in the first non-trivial forward step. This limits the algorithm to have at least two observations and prompts some post scaling after the loop has ended.

Fuse backward and update step. The end products needed are π,*a** and *b**, which are reliant on Γ and Ξ. If one inspects carefully, one can see that only two βs (one for the current step and one for the previous step) are needed to compute the next β. This amount can be dealt with without necessarily thinking about saving them in a $N \times T$ big matrix, if the β gets used right away.

$$\beta_i(t) = \sum_{j=1}^{N}\beta_j(t+1)a_{ij}b_j(y_{t+1})$$

$$\gamma_i(t)^* = \alpha_i(t)\beta_i(t)$$

$$\xi_{ij}(t)^* = \alpha_i(t)a_{ij}\beta_j(t+1)b_j(y_{t+1})$$

Looking at this, one can reuse most of the results since:

$$\gamma_i(t)^* = \sum_{j=1}^{N}\xi_{ij}^*(t)$$

$$\beta_i(t) = \frac{\sum_{j=1}^{N}\xi_{ij}(t)}{\alpha_i(t)}$$

Correct y_t. The most obvious way to compute $1_{y_t=v_k}$ used for *b** is to iterate over all *K* different signals, but since y_t is in this implementation only one known signal, the corresponding $\gamma_i(t)$ can be added in the correct *b** place instead of iterating over all the possible different signals.

Pre-computing ab. Most of the final loops iterate over time and the next step is dependant on the previous step, which hinders pre-computing. One exception to that is $a_{ij} * b_i(y_t)$ in the fused step. The pre-computed matrix has size KN^2 but gets accessed N^2T times, which is a big gain on memory accesses and also flop (if according to the assumption $K < T$).

Access patterns. To enable better spatial locality it is important to analyze the code and store matrices in the correct order, in which the algorithm accesses the data.

$\alpha_j(t)$ is used in the forward step to compute $\alpha_i(t+1)$, where it has to be fetched in state order, the same as in the second step. ***a*** is needed in first-state order for the forward step, but needed in second-state order for creating *ab*. This means that a **transpose** is needed for ordering *a* after the forward step to utilize spatial locality of *a*. This makes sense because the paid $O(N^2)$ to transpose reduces the loads of the matrix to one fourth in $O(KN^2)$. This is also the only time where **blocking** is used, as most other operations over matrices are dependent on time and matrices might change while the first block is frozen. ***b*** gets accessed in state order for the forward step and also in the second step. ***ab*** gets accessed only for the second step, where it is used in state order. All the other matrices are either vectors, could be reduced to vectors or one of the above matrices during the reordering step.

Theoretical results. The analysis assumes $K \approx N < T$ and uses the same way of counting as in the background chapter.

With this reordering step, the flops get lowered from the standard $3KNT + 10N^2T$ to $KN^2 + 6N^2T$, which is a

asymptotical save of about $\frac{13}{6}$.
The standard $2KNT + 14N^2T$ memory accesses get reduced to $3KN^2 + 6N^2T$ in the reordered version. This means a reduction of roughly $\frac{8}{3}$.
The last metric provided is the working set, which was $TN + 4T + N + K$ in the standard version and is new $2N^2 + 6N$, this can be a big save, since the new working set is not reliant on the big *T*.

3.3. Vectorization

In order to be able to apply vectorization the reordered code had to be unrolled. The algorithm can not be unrolled across different timesteps because the result of the current timestep depends on the previous timestep. Therefore it is only possible to unroll across states and observables, what our algorithm did to the fullest to fill processor pipelines longer and enable better instruction level parallelism. The stepsize is 4 as experiments with other stepsizes did not improve the performance. As a result, the number of states and observables in the model has to be dividable by 4.
For the vectorization, the FMA instruction set was used. Best effort was made to make the computations as independent as possible. Unfortunately, some reductions in the forward and update step had to be used. They are especially costly because they need permuting and shuffling.

3.4. BLAS version

The BLAS version tries to apply BLAS[5] functions as much as possible to the reordered code without the transposes. BLAS 1 was useful for scaling α and β and to compute the dot product of the transition matrix and α in the forward step. Other BLAS modules could hardly been used as the reordered code is not optimized to incorporate BLAS functions. Probably there are better implementations which benefit more from BLAS. The current reordered version depends a lot on element-wise vector multiplications, which is not included in BLAS, but one could use the Intel math kernel library.[6]

4. EXPERIMENTAL RESULTS

4.1. General

Plots. Performance plots, that analyze the impact of each of the three parameters (hidden states, different observables and observations), a section about environmental comparisons (machine, compiler, flags), and a also time plot, that analyzes the different algorithm versions with each other are presented in this chapter next to a roofline plot and a Cachegrind table that compares different versions.
N-Plots. To reduce the amount of variables per plot, there is a meta variable (*N*) introduced, which is on the X-axis of many plots. N is the amount of hidden states, of different observables and the root of signals received, since by assumption, this variable is biggest. E.g. N=16, hidden States=16, different observables=16, received signals=256.
Code version names. To save space on the plots, the different code versions have different abbreviations:

- stb=stable (base)
- cop=C optimized
- reo=reordered
- vec=vectorized
- bla=optimized with BLAS
- umd=alternative library (umdhmm)

System specs. The measurements have been taken on different machines running Linux Ubuntu, but for each plot, the measurements come from the same system unless indicated otherwise. The system is annotated under each figure. The input matrices get generated before the run of the algorithm and, to make cold cache measurements possible, the cache gets flushed.
System 1 used has a Intel core i5-3210M(Ivy Bridge, 2.5GHz, 3MB cache, 2 flops/cycle peak) [7] and uses gcc version 7.5.0 and icc version 19.1.1.219.20200306.
System 2 has an Intel core i7-4790K [8](Haswell, 4GHz, 8MB cache, 4 flops/cycle (scalar), 16 flops/cycle (vector) peak) and uses gcc version 9.3.0, icc version 19.0.5.218 and Valgrind-3.15.0.
Flop count/memory access count. The counts are the same as the ones presented in the optimizations subsection "reordered" and the paragraph "theoretical results". The counting has been done for the stable version and the reordered version, since the counts have only changed by a relatively small amount of constant steps between other versions.
Comparison with other libraries. Because of the fact that timing was implemented such that each loop iteration is measured, it is hard to compare this implementation to a library, since doing the same in such a library is usually not easy. Because of its straightforward implementation, the umdhmm library [9] offered a way to compare, by manually implementing the same benchmarking structure into its code.

4.2. Impact of different parameters

There were some measurements collected to study the effect of the three parameters *hidden states, different observables* and *observations T*. The behaviour of the stable version and the C-optimized version were as expected comparable. The same holds for the comparison of the reordered version to the vectorized version. Because of that only the plots for the C-optimized and vectorized versions are showed to keep it tidier. Everything was compiled with gcc and the -O2 optimization flag. In addition the vectorized version used the -mfma flag.

Impact of the number of observables. The first analysis

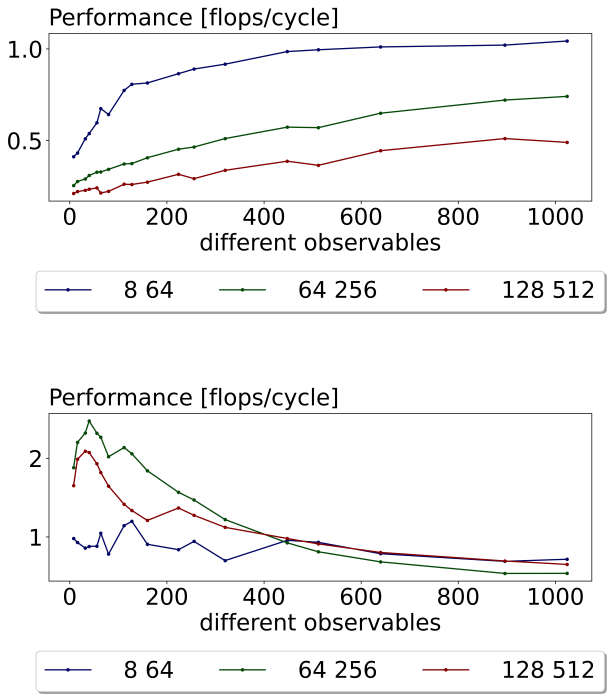

Fig. 1. Impact of the number of different observables to the performance of C-optimized (top) and vectorized (bottom) on system 2 with gcc flags -O2. The lines correspond to hidden states, number of observations, e.g. the blue curve corresponds to 8 hidden states and 64 observations.

(Figure 1) focuses on the impact of the number of different observables. The performance of the C-optimized version increases with the amount of different observables. On the other hand, the performance of the vectorized version decreases with the number of observables. For the medium and the large case of the vectorized version one can see two kinks. The first is around 64, which occurs because the L1 cache will be full and loading from the slower L2 cache is needed. The second decrease in performance is around 512, where the L2 cache will be full and loading from the slower L3 cache is needed. The performance of the small case is more or less constant.

Impact of the number of hidden states. Both versions of the hidden states plot (Figure 2) show a similar behaviour. The biggest difference is the increase in performance for small numbers of hidden states for the vectorized version instead of a decrease for the C-optimized version. The increase in performance is linked to the transposition of the transition matrix. For a small number of hidden states this is not beneficial but for a higher number of hidden states this gets advantageous. The first significant drop is around

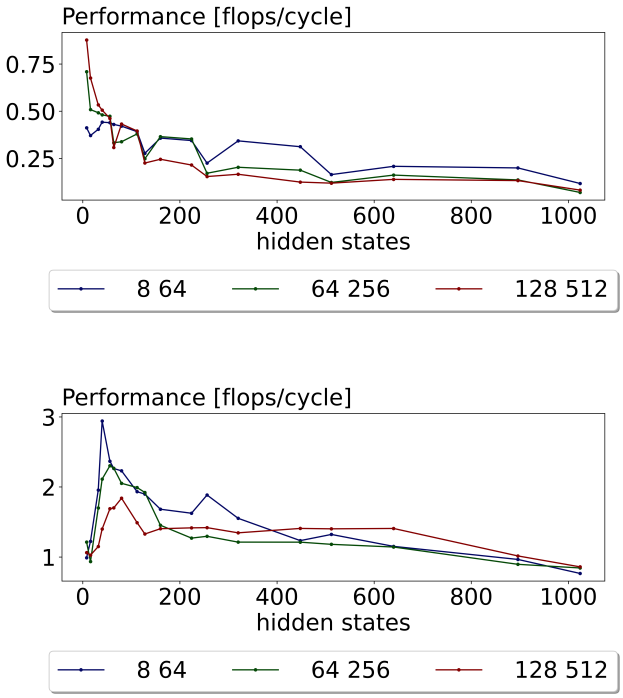

Fig. 2. Impact of the number of hidden states to the performance of C-optimized (top) and vectorized (bottom) on system 2 with gcc flags -O2. The lines correspond to different observables, number of observations, e.g. the blue curve corresponds to 8 different observables and 64 observations.

64 and corresponds to the size of the L1 cache. The next drop is around 512 and corresponds to the L2 cache.

Impact of number of observations. The two plots in Figure 3 show the big advantage of the reordering. There is no matrix anymore that scales with the number of observations T. Therefore you can see a flat line for the vectorized version as the number of observations increases. In contrast, the performance of the C-optimized version decreases as matrices that scale with T are needed.

4.3. Environment comparisons

Machine comparison. The two systems of the machine comparison (Figure 4) behave as expected in most cases parallel to each other, where the slower system 1 trails system 2 on all the versions that system 1, which does not support vector instructions, can run. System 1 is at around 50% much closer to system 2's 30% peak performance since it can not execute FMA instructions as well. The only thing that catches the eye is that the last measurement of the reordered version on system 1 is higher than the previous points, where it seemed to have plateaued. This rise of about

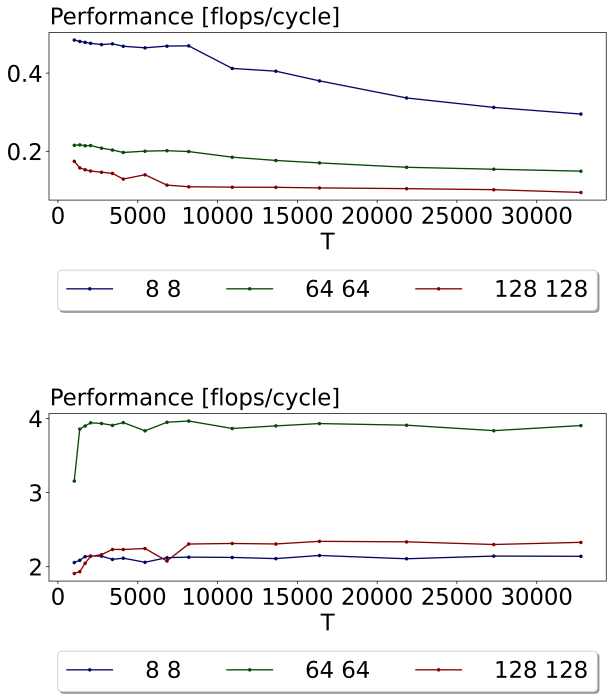

Fig. 3. Impact of the number of observations T to the performance of C-optimized (top) and vectorized (bottom) on system 2 with gcc flags -O2. The lines correspond to hidden states, different observables, e.g. the blue curve corresponds to 8 hidden states and 8 different observables.

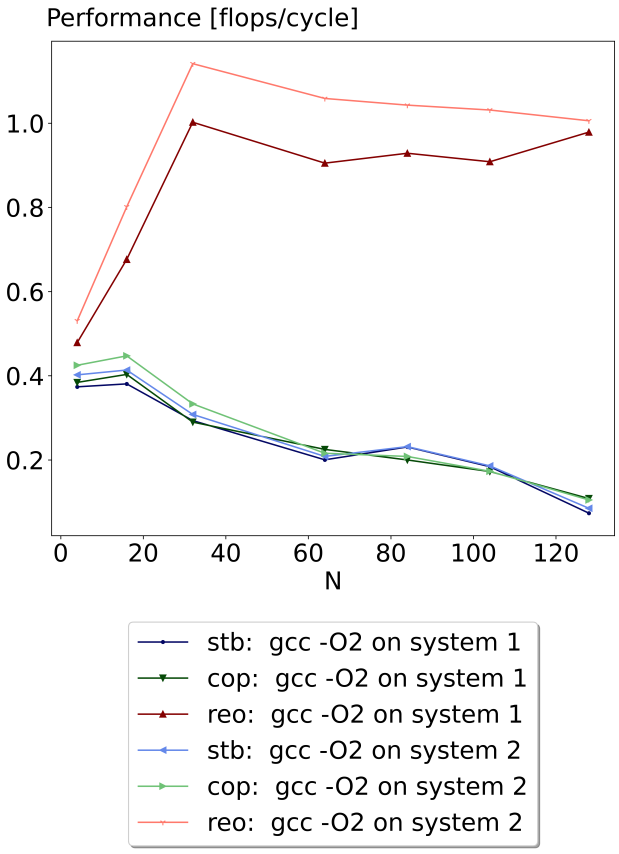

Fig. 4. Performance comparison of different versions on different systems with gcc.

0.1 is small enough that natural testing variation can be a factor.

Compiler comparison. The compiler plot (Figure 5) shows again a performance comparison between the different versions and that the difference between gcc and icc is small in comparison. Zooming in on the different versions, there is no difference to make out for the stable and C-optimized version. More interestingly, icc is always noticeably better than gcc in the reordered version without vectorization. For the vectorized version the performance measurement at N=16 is lower for gcc and higher for icc, unlike the bigger values, where gcc is slightly better. The reason why icc is better than gcc for the small N might due to the same effects of the sequential reordered version, which outweigh the vectorization gain.

Flags comparison. Flags only have a small impact on the performance of the code as can be seen in the flags plot (Figure 6) with the big exception being the change from gcc -O2 to -O3 for the non-vectorized reordered version, that grows faster and also plateaus at a higher performance level. This gain could be because the -O3 flag may detect transposing and make it faster or does better pre-fetching on the long loops.

4.4. Comparison of different implementations

Roofline plot. In this roofline plot (Figure 7), one can see that the computation is memory bound for the stable as well as for the C-optimized version, since the operational intensity does not change between these two. The reordered and vectorized versions manage to escape this memory bound and are compute bound; again the operational intensity does not change between them.

While the C-optimized version does not improve performance drastically compared to the stable version, the vectorized version improves performance by a large factor compared to the reordered version. This happens because the C optimizations only increase performance slightly and the SIMD instructions have a much larger impact. Notice also that the reordered version improves performance compared to the previous versions.

Cachegrind results. To open the black-box of the algorithm a bit, one can have a look at the Cachegrind compar-

	Instruction reads (mil)	Cache accesses (mil)	% Cache misses	Branches (mil)	% Branch misses
Stable	88933(100%)	34159(100%)	5,5	10201(100%)	1.06
C-Optimized	75088(84%)	25464(75%)	7,5	10201(100%)	1.06
Reordered	29627(33%)	10446(30%)	**0,0007**	3463(34%)	1.56
Vectorized	**3894(4%)**	4633(14%)	**0,0014**	233(3%)	6.20

Table 1. Cachegrind output comparison between code versions for N=64 on system 2 with gcc and flags -O1-mfma.

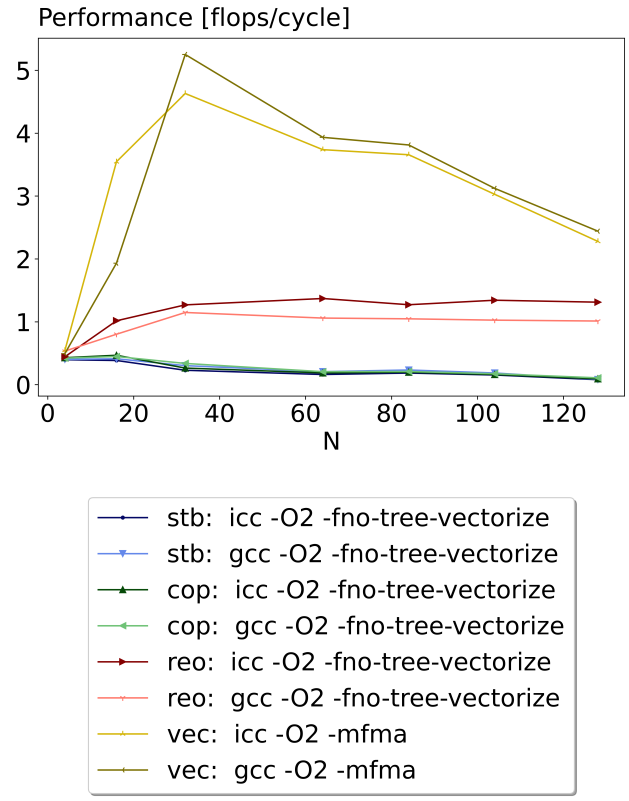

Fig. 5. Performance comparison of compilers between icc and gcc on system 2.

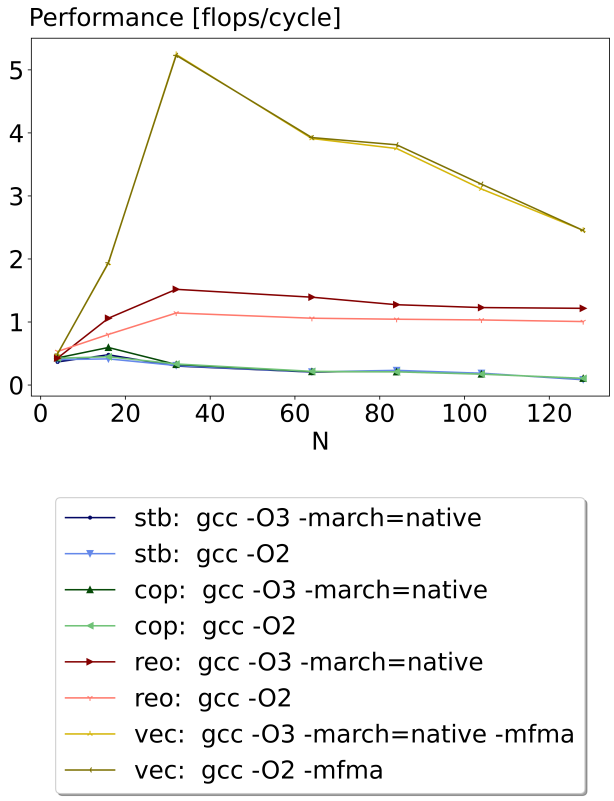

Fig. 6. Performance comparison of different gcc flags on system 2.

ison (Table 1), where the different code versions compiled with gcc and flags -O1 get analyzed on system 2 for the parameter N=64 with warm cache. Cache accesses are from data reads and data writes. Cache misses are the combined last level cache misses of data reads and data writes compared to the overall accesses. The amount of instruction read misses is minuscule. The percentage in the brackets is the comparison to the stable version.

In this analysis, one can see that the stable version and the C-optimized version are comparable in most metrics, where the C-optimized version uses a bit less instructions and cache accesses, but one can clearly see by the amount of branches that they go through the same order of steps; C-optimized has shortened each iteration of the loops, but did not change the amount of loop iterations. The absolute amount of cache misses is the same, but since the stable version has more cache accesses they are a lower percentage.

Looking at the reordered version, one can see that the amount of loop iterations (branches) has been reduced to about a third of the stable version, and the amount of cache accesses and instruction reads follow suit. One can deduce from this that the amount of work per iteration is similar, but the amount of iterations got reduced by a lot. What is even more important for a memory-bound algorithm is that the cache misses got reduced greatly by a factor of about a thousand. This comes in great parts from the removal of

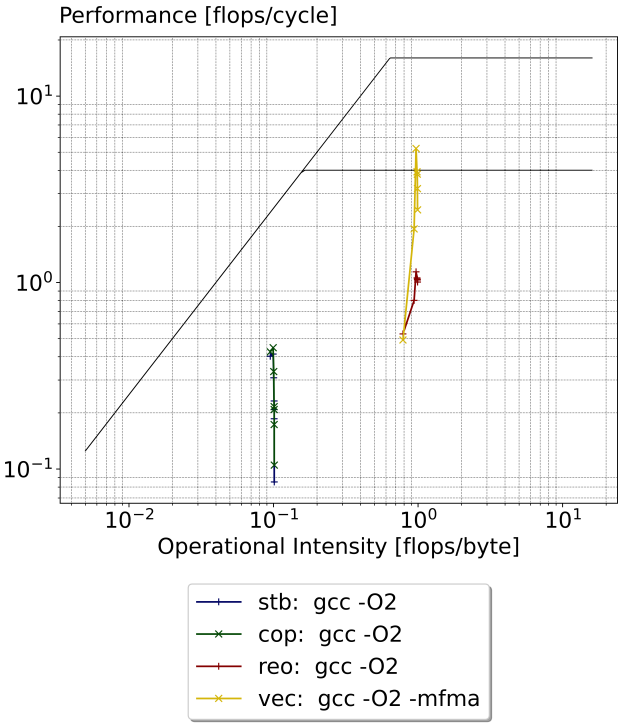

Fig. 7. Roofline plot for the different versions compiled with gcc -O2 on system 2.

writing each β to only use the two that are used right now, which can always stay in cache. For the case of N=64 the writes to memory are minuscule.

After numerous close inspections of the code and the testing infrastructure the vectorized version is still at the unintuitive result of half the cache accesses as the reordered version, even though they use the same code as base. This is also reflected in the amount of branches, which is half from the expected factor of a fourth(since each loop iteration goes over 4 states), the same as the amount of instructions, yet the amount of missed branches is as expected four times higher than from the reordered version. A possible explanation for the factor 2 could be that the vectorized version handles the transposing differently; either it does not do the transposing at all or already when creating the matrices in the first place.

Time plot. As final plot, the cycles plot (Figure 8) gets presented as it gives the most information to the run-time sensitive end-user. There it can be observed that the vectorized, BLAS and reordered version are in a league of their own, where vectorized is best followed by the reordered version until the BLAS version. This could be because all of them have the reordered code as base. Interestingly, the BLAS version is slower than the reordered version without libraries, this can be explained by the fact that BLAS1 is

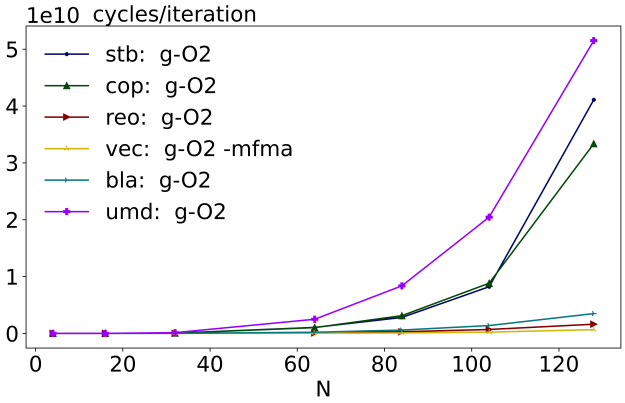

Fig. 8. Cycles comparison between different implementations per iteration on system 2.

not as big of a boost as the other modules of BLAS are and each time BLAS is called, a new function gets called, which creates overhead.

Following, it becomes apparent that the other versions grow exponentially with N, like the observed signals (T). As expected, the C-optimized version grows slowest, followed by the stable version and umdhmm trails the competition significantly.

5. CONCLUSIONS

In this paper the Baum-Welch algorithm got analyzed and optimized, first for stability and then also for speed. Next to an inspection on the impact of the different parameters, the most interesting insight came by measuring the peak speed up of 10X which got achieved by fusing together the backward and update step. The vectorized version lowered the amount of instructions to about a twentieth of the stable version. Also the working set is no more dependent on the amount of observations and the overall execution time could get lowered greatly as illustrated in Figure 8.

Other ways to explore the Baum-Welch algorithm would be to reorder the instructions, so it suits the BLAS library best and compare the results of that library with the manually vectorized version. Another possible exploration path would be to generalize the algorithm to allow more than one signal at a timestep or multiple observation sequences.

6. CONTRIBUTIONS OF TEAM MEMBERS

Luca. initially implemented the forward step, did the stability improvements and simple C optimizations, afterwards converted the outlined reordering steps into code, was part of the vectorization process and oversaw the BLAS version.

Next to coding also did the profiling of the stable version, finalized the plotting and caught numerous bugs.
Jannik. introduced many theoretical optimizations for the reordered version, which then were implemented into the reordered version. Did the inlining for all versions and started with the BLAS version. Oversaw the Valgrind analysis and laid the groundwork for plotting, next to profiling the reordered version.
Jan. implemented the finishing criteria and implemented a version with blocking in the forward step. Unrolled the reordered version before starting with the vectorization and helped with the vectorization of this version.
All of us. Read, validated and corrected code changes by others and discussed the next steps and future ideas for the project. Also, everybody ran some tests.

7. REFERENCES

[1] Leonard E. Baum and Ted Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *Ann. Math. Statist.*, vol. 37, no. 6, pp. 1554–1563, 12 1966.

[2] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[3] Byung-Jun Yoon, "Hidden markov models and their applications in biological sequence analysis," *Current genomics*, vol. 10, pp. 402–15, 09 2009.

[4] István Miklós and Irmtraud Meyer, "A linear memory algorithm for baum-welch training," *BMC bioinformatics*, vol. 6, pp. 231, 02 2005.

[5] Intel Corporation, "Blas and sparse blas routines," 2020 (Accessed 4 June 2020), https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top/blas-and-sparse-blas-routines.html.

[6] Intel Corporation, "Intel math kernel library," 2020 (Accessed 4 June 2020), https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[7] Intel Corporation, "Intel core i5-3210m processor," 2020 (Accessed 4 June 2020), https://ark.intel.com/content/www/us/en/ark/products/67355/intel-core-i5-3210m-processor-3m-cache-up-to-3-10-ghz-rpga.html.

[8] Intel Corporation, "Intel core i7-4790k processor," 2020 (Accessed 6 June 2020), https://ark.intel.com/content/www/us/en/ark/products/80807/intel-core-i7-4790k-processor-8m-cache-up-to-4-html?wapkw=i7-4790K.

[9] Tapas Kanungo, "Umdhmm," 2020 (Accessed 10 June 2020), http://www.kanungo.com/software/software.html#umdhmm.