OPTIMIZED IMPLEMENTATION OF THE BAUM-WELCH ALGORITHM

Olivier Bitter, Karel Jílek, Jonas Mohler, Ylli Muhadri

Department of Computer Science
ETH Zürich
Zürich, Switzerland
June 12, 2020

ABSTRACT

The Baum-Welch algorithm consistently finds use in applications across disciplines. As these tend to be resource-intensive, recent years have seen a fair share of research on improvements. Previous work has focused primarily on the theoretical bounds of Baum-Welch, and its adaption to suite large-scale compute clusters.

We, on the other hand, strive to find the gains in performance and speed we can get from profiling and optimizing a straightforward C implementation on a single core from Intel microarchitectures supporting AVX2. We see an opportunity for further speedup leveraging a trade-off between memory and computation, and from there on, continue exploring both directions of these optimizations.

We find that we can increase performance by up to a factor of 5.83, and reach 27 % of theoretical peak performance with our most advanced efforts, surpassing any comparable implementation known to us. Moreover, the best runtime we could achieve amounted to only 20 % of the baseline.

1. INTRODUCTION

Motivation. The theoretical foundations of Hidden Markov models (HMMs) were established close to a century ago. In the past 30 years, they have increasingly gained popularity across scientific disciplines.

HMMs model relations between a set of observable and hidden states of a system. Especially the task of learning the parameters of an HMM, such as the probabilities for transitioning between states, has found various applications in recent years. Baum-Welch, a special case of the expectation-maximization algorithm, implements the learning task of HMMs, and after its creation quickly found use in the field of speech recognition.[1][2]

In the meantime it has also found use in numerous other fields, cryptanalysis being one of them, recovering phrases from encrypted VoIP[1] calls, or in biology, predicting entire structures of human genes from a known subset.[3][4]

These applications typically work on large data sets and may require real-time processing. Obtaining reasonably fast implementations of Baum-Welch is thus critical for the dissemination of these technologies.

Related work. Theoretical work on Baum-Welch has previously provided modifications like fast state selection, exhibiting better asymptotic complexity. These adaptions may, however, come with decreases in precision. [5][6]

Other improvements of recent years have brought forth linear memory and parallel versions of Baum-Welch, the latter targeting deployment on compute clusters. [7][8]

Libraries for HMMs are only sparsely available across languages and are often not well documented and unclear as to what version of Baum-Welch they are implementing, if any, and what performance they can achieve.

We, in contrast, focus on producing a fast version of the original Baum-Welch algorithm in C, observing and reporting the performances we can achieve using combinations of standard C optimization methods, improved memory access locality, and vectorization using Intel's AVX2 instruction set. We optimize the algorithm to perform as well as possible on a single core of an Intel Skylake processor.

2. BACKGROUND: HMM AND BAUM-WELCH

HMMs are generative models in which the joint distribution of the observation and hidden states is modeled. The hidden state is assumed to be a Markov process while the observation is a probabilistic function of the hidden state. [1]

Elements of an HMM. HMMs are specified by the following parameters [2]:

1. N, the number of states in the model. We denote the individual states as $S = \{S_1, S_2, \ldots, S_N\}$, and the state at time *t* as q_t.

2. M, the number of distinct observation symbols per state, i.e. the discrete alphabet size. We denote the set of the symbols as $V = \{v_1, v_2, \ldots, v_M\}$ and the

[1] Voice over IP

[2] Notation is based on [1] and [9].

sequence of observed symbols as $O = O_1 O_2 \dots O_T$ of length T.

3. The state transition probability distribution $A = \{a_{ij}\}$, where $a_{ij} = P[q_{t+1} = S_j | q_t = S_i], \quad 1 \leq i, j \leq N$.

4. The observation symbol probability distribution in state j, $B = \{b_j(k)\}$, where $b_j(k) = P[O_t = v_k | q_t = S_j]$, $1 \leq j \leq N, 1 \leq k \leq M$.

5. The initial state distribution $\pi = \{\pi_i\}$, where $\pi_i = P[q_1 = S_i], \quad 1 \leq i \leq N$.

We represent the parameters of the model compactly as $\lambda = (A, B, \pi)$. We will focus on ergodic, discrete HMMs.

Baum-Welch. The Baum-Welch algorithm solves one of the three main HMM problems, namely finding the model parameters $\lambda = (A, B, \pi)$ that maximize $P(O|\lambda)$. It does so by iteratively updating an initial $\lambda = (A, B, \pi)$ such that $P(O|\lambda)$ is locally maximized. First, we define two important variables for the Baum-Welch algorithm:

- Forward variables

$$\alpha_t(i) = P(O_1 O_2 \dots O_t, q_t = S_i | \lambda)$$

- Backward variables

$$\beta_t(i) = P(O_{t+1} \dots O_T | q_t = S_i, \lambda)$$

It is known that a straightforward computation of these variables is numerically unstable, i.e. the dynamic range will exceed the precision range of modern processors[1]. Scaling with a factor that depends only on *t* or doing the computations in log-space are two possible solutions to the problem. We will directly present the scaled computation of the forward and backward variables below.

Initialization	Induction
$\alpha_1(i) = \pi_i b_i(O_1)$	$\ddot{\alpha}_t(i) = \sum_{j=1}^{N} \ddot{\alpha}_{t-1}(j) a_{ji} b_i(O_t)$
$\ddot{\alpha}_1(i) = \alpha_1(i)$	$c_t = \frac{1}{\sum_{i=1}^{N} \ddot{\alpha}_t(i)}$
$c_1 = \frac{1}{\sum_{i=1}^{N} \ddot{\alpha}_1(i)}$	$\hat{\alpha}_t(i) = c_t \ddot{\alpha}_t(i)$
$\hat{\alpha}_1(i) = c_1 \ddot{\alpha}_1(i)$	

$\ddot{\beta}_T(i) = 1$	$\ddot{\beta}_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j)$
$\hat{\beta}_T(i) = c_T \ddot{\beta}_T(i)$	$\hat{\beta}_t(i) = c_t \ddot{\beta}_t(i)$

After computing the forward and backward variables, we can update the model parameters $\lambda = (A, B, \pi)$ as follows:

$$\bar{\pi}_i = \hat{\alpha}_1(i)\hat{\beta}_1(i)/c_1$$

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \hat{\alpha}_t(i) a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j)}{\sum_{t=1}^{T-1} \hat{\alpha}_t(i)\hat{\beta}_t(i)/c_t}$$

$$\bar{b}_j(k) = \frac{\sum_{t=1, O_t = v_k}^{T} \hat{\alpha}_t(j)\hat{\beta}_t(j)/c_t}{\sum_{t=1}^{T} \hat{\alpha}_t(j)\hat{\beta}_t(j)/c_t}$$

The algorithm is usually run until $P(O|\lambda)$ does not significantly improve anymore, or a reasonable number of iterations is reached. In our implementation, the algorithm stops after a fixed number of iterations to make the stopping criterion deterministic and the experiments comparable.

Data: Observation $O = O_1 O_2 \dots O_T$ of length T, Initial $\lambda = (A, B, \pi)$, Number of iterations *max_it*
Result: Locally optimal $\lambda^* = (A, B, \pi)$
for $it = 1 \dots max\_it$ **do**
  compute forward variables;
  compute backward variables;
  update π;
  update A;
  update B;
end

Algorithm 1: Baum-Welch Algorithm

Cost Analysis. We decided to use the standard flop count as a measure for the cost C:

$$C = N_{add} + N_{mul} + N_{div}$$

The exact cost for a fixed number of iterations k of the straightforward C implementations is as follows:

$$C(N, M, T, k) = k \cdot (13N^2(T-1) + 3NMT + N^2 + 6NT + NM + 2N + T)$$

3. PROPOSED METHOD

Starting with the baseline version, a straightforward implementation of the algorithm described in section 2, we first identify areas with potential for improvement using the baseline profile and roofline analysis. Then, we apply the corresponding optimizations described below to achieve as optimized standard C code as possible. After that, we apply AVX2 on the best standard C code to fully utilize the CPU's potential.

As shown in Algorithm 1, one iteration of the Baum-Welch algorithm can be divided into five sections. Those

depend on each other in the presented order, so it is neither possible to change their order nor to interleave them. However, it might be possible to reuse some auxiliary values across sections. The goal will thus be to optimize each section separately while finding values that can be reused.

Baseline profile. First, we analyze how the baseline version is performing, as such findings can help us determine which optimizations to apply.

The baseline profile in figure 1 reveals that almost equal amount of time is spent in all of the sections except one, which consists of just a single loop; therefore, its low runtime is expected. There is no section consuming significantly more time than the others.

The roofline plot in figure 2 shows that the baseline version is significantly compute-bound. Therefore, if we manage to replace some computations with memory movements, i.e. storing some intermediate results and reuse them, we expect it to result in a better runtime.

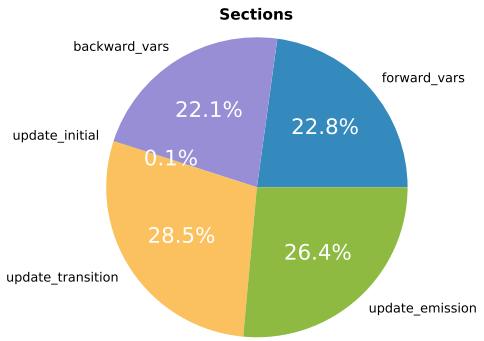

Fig. 1. Profile of the baseline version for $N = 64$, $M = 64$ and $T = 8192$.

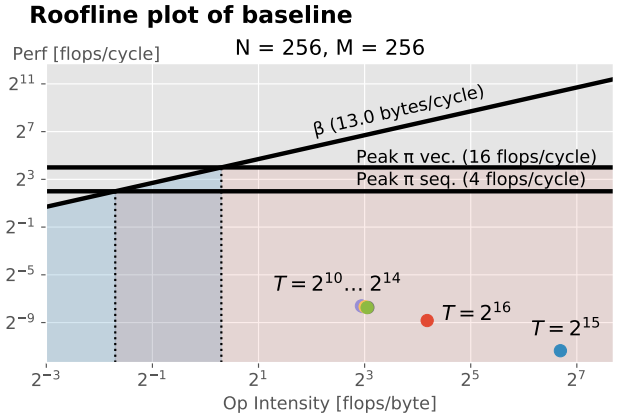

Fig. 2. Roofline plot of the baseline version.

Spatial locality. To improve spatial locality (and therefore cache miss rate), we found that it might be worth it to store the forward variables, backward variables, and observation symbol probabilities in transposed order. Changing the loop order falls into the same category, trying to ensure as many sequential accesses to arrays as possible.

Loop unrolling. By itself, loop unrolling does not do much. However, it allows us to increase instruction level parallelism using separate accumulators for reduce operations, breaking the sequential dependency chain between operations. Further, unrolling by a factor which is a multiple of 4 makes the AVX2 vectorization more straightforward, as each AVX2 vector fits exactly 4 double-precision floating-point numbers. Figure 3 shows an example directly from one of the optimized versions of the code.

```
for(int i = 0; i < N; i+=4) {
    double f0 = PI[i]*B[i*M + o];
    double f1 = PI[i+1]*B[(i+1)*M + o];
    double f2 = PI[i+2]*B[(i+2)*M + o];
    double f3 = PI[i+3]*B[(i+3)*M + o];
    s0 += f0;
    s1 += f1;
    s2 += f2;
    s3 += f3;
}
double s = s0 + s1 + s2 + s3;
```

Fig. 3. An example of loop unrolling by 4 (`opt_stdc_mem.cpp`, line 72).

Inverting division. Compared to multiplication, division is a costly operation. On Skylake, its latency is up to 14 cycles and its throughput is $\frac{1}{8}$ instructions per cycle, while multiplication has latency and throughput four cycles and two instructions per cycle respectively. [10] Therefore, for some set $S \subset \mathbb{R}$, it might be worth not computing $S_{div} = \{x/C \mid x \in S\}$, but instead first compute $c := 1/C$ and then $S_{mul} = \{x \cdot c \mid x \in S\}$. This way, there will be just one division instead of the original $|S|$. Figure 4 shows an example. Unlike in exact arithmetic, using floating-point operations, S_{div}'s and S_{mul}'s corresponding elements might slightly differ. However, it should not matter much, since floating-point arithmetic is already inexact.

```
double c = 1/C;
for(int i = 0; i < N; ++i){
   a[i] *= c;   // same as a[i] /= C
}
```

Fig. 4. An example of inverted division.

Scalar replacement. Updating a memory location in the worst case means a cache miss, loading the data from main memory to cache, performing the actual update, and

soon after, an eviction and a write-back to main memory. If the same memory location is updated repeatedly, it might result in a huge performance loss. A standard workaround is to keep the value in a register, and only once all the updates are done, write it back to the memory. This results in at most two memory movements for N operations, compared to the original N. Figure 5 shows an example.

```
double scalar = a[0];
for(int i = 0; i < N; ++i){
   scalar += b[i];
}
a[0] = scalar;
```

Fig. 5. An example of scalar replacement.

Reusing of values. The denominators computed in the update transition and update emission sections can be reused with some small modifications. This reduces the asymptotic time complexity of the algorithm. However, it creates a lot of additional memory movements, which may result in an overall slowdown of the computation. However, as the roofline plot in figure 2 shows, the baseline version is compute-bound. Thus, reducing the number of computations while increasing memory movements should result in an overall better runtime. In the experiments, we call the family of optimizations including this idea the *memory versions*[3], as opposed to *base versions*.

Applying AVX2. All the optimizations mentioned above form the best standard C code we could achieve. Choosing the unrolling factors four and eight made applying AVX2 generally straightforward: the whole vectorization avoided expensive gather operations, using shuffles only in a few instances. The theoretical speedup after applying AVX2 is up to 4x. This is also the best we can achieve without using parallel or distributed computing.

Algorithm constraints. Due to the chosen unrolling factors of four and eight, there is a constraint on the input size to be divisible by these factors. For production code, one would have to take care of leftover iterations when the input size is not divisible by these factors.

4. RESULTS

We present the benchmarking pipeline used for all experiments. Next, we discuss the impact of different compilers and levels of compiler optimizations, which lead us to determine the final setup for benchmarking. Finally, we analyze the performance of the optimizations. We have two sets of optimizations, the base and the memory version. We discuss the performances of both versions in isolation and compare their runtimes in the last paragraph.

[3]Since it uses additional memory.

Experiment setup. All experiments were run on an Intel Whiskey Lake processor, an optimization of the Skylake architecture, which implements instruction sets up to and including AVX2. Table 1 shows a summary of benchmarking platform.

Operating System:	Ubuntu 20.04 LTS
CPU Name:	Intel Whiskey Lake
CPU Number:	i5-8265U
Base frequency	1.8 GHz
Max frequency:	3.9 GHz
Turbo Boost:	Disabled
L1 cache:	128 KiB
L2 cache:	1 MiB
L2 cache:	1 MiB
Peak Performance:	16 Flops/Cycle

Table 1. Summary of the platform used for the experiments.

During benchmarking, we made sure to isolate a single core for running the experiments, preventing the operating system from scheduling processes on this core. Furthermore, we prevented all interrupt requests that are controllable from the user space to be served by that core. We then manually schedule our processes on that core. We use the PAPI library for measuring and reporting hardware events such as instruction counters and memory accesses. [11] As hardware counters can produce meaningless results when using Intel® TurboBoost technology, we made sure to always disable it while running experiments. Every experiment involves the following parameters:

- An optimized implementation to benchmark, written in C.
- A compiler to compile the source code from.
- A set of compiler flags, compatible with the provided compiler.
- An input range for the parameter N, while keeping M and T fixed.
- An input range for the parameter M, while keeping N and T fixed.
- An input range for the parameter T, while keeping N and M fixed.

Our benchmarking pipeline compiles the provided implementation using the provided compiler and flags. Before running the compiled executable, over the three specified ranges, we first validate whether the program produces valid results by running it on a small input, and comparing the output to a manually validated implementation. Upon successful validation, we run the experiment over the three

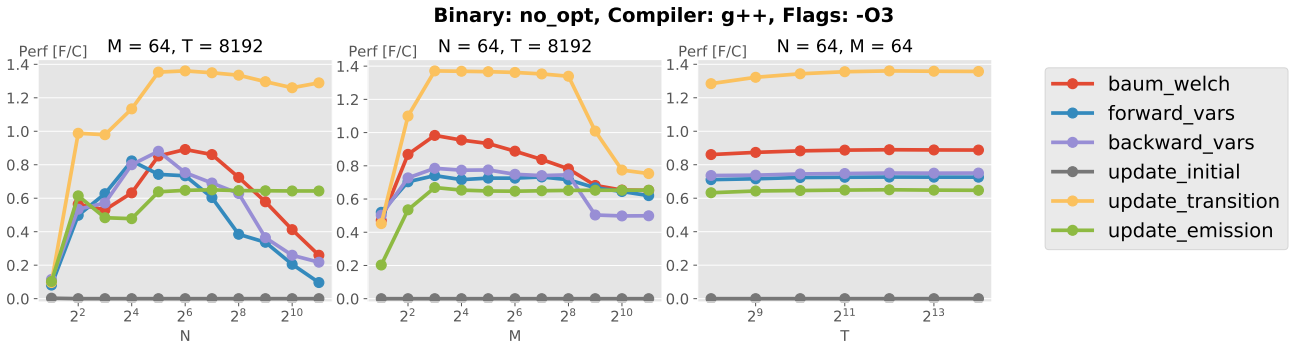

Fig. 6. Comparison of the performance of the individual sections of the unoptimized base implementation. Each plot visualizes the effect of varying one of the three input dimensions: number of hidden states (N), observation alphabet size (M) and observation sequence length (T).

specified ranges and store the measurements in a `.csv` file. During benchmarking, we count the CPU cycles, the number of floating-point operations, and the last level cache (LLC) misses. This data enables us to generate runtime, performance, and roofline plots, which we will discuss later. Throughout this section we refer to the shown plots as *N-plot*, *M-plot* or *T-plot* depending on the variable on the x-axis. We take these measurements for the entire algorithm and all individual sections to get insights on how the optimizations impact different parts of the algorithm and identify performance bottlenecks. Figure 6 shows the output of an experiment run on the unoptimized baseline. As the input size grows quadratically with N but only linearly with M and T, the N-plots turn out to have the biggest impact on performance. We thus limit the focus mostly on the N-plots for the analysis of the remaining experiments in this section.

Compiler experiments. We compare the icc compiler from Intel with the gcc compiler from GNU and experiment with different flags on both compilers to see how much performance we can gain from using compiler level optimizations only. To this end, we ran experiments on our baseline implementation `base_no_opt`. The N-plots of said runs can be seen in figure 7.

Both compilers yield a stable performance slightly below 0.2 F/C (flops/cycle) by explicitly turning off all compiler optimizations using `-O0`. We see that using gcc results in an increase in performance with higher optimization levels. The flag `-O1` results in a 2x speedup and `-O3` even in a 4x speedup. We observe some unexpected behavior measuring the same compiler flags in icc. Using `-O1` results in a better performance than with `-O3`. Due to this anomaly, we decided to use gcc with `-O3`. Additionally, we make sure to enable AVX2 in the experiments presented from here on, resulting in the final setting: gcc `-O3 -march=native`.

Base version optimizations. The performance N-plots of the base version optimizations can be seen in figure 8. The implementation `base_no_opt` is the straightforward

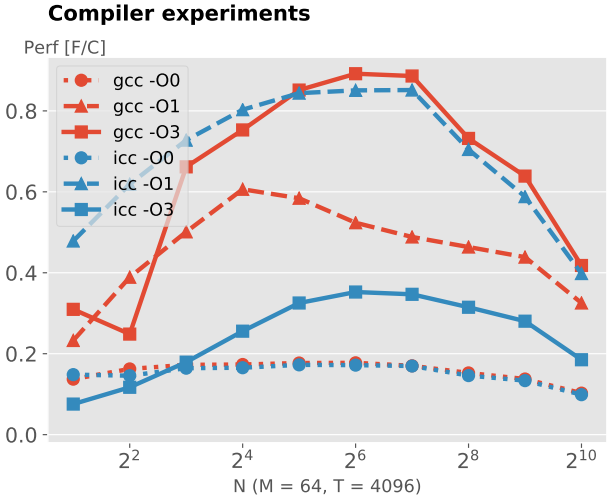

Fig. 7. Comparison of gcc (red) and icc (blue) with different levels of compiler optimizations.

C implementation without any optimizations. The performance starts slightly above 0.5 F/C and peaks at 0.75 F/C. The implementation `base_stdc` contains standard C optimizations without AVX2. The performances range from 0.7 F/C to 1.2 F/C and we reach a speedup of up to 1.62 as compared to the unoptimized version.

The vectorized `base_avx_tp` achieves the best performance. This optimization imposes a minimum constraint on the input size to enable further unrolling for vectorization, which is why data points for small inputs are missing for this line. By adding vectorization, we gain a speedup of up to 5.83 in the optimization `base_avx_tp`, which equates to 27 % of peak performance. A summary of all speedup comparisons is listed in table 2. The performance for `base_avx_tp` increases up to $N = 2^5$ and reaches a performance of 4.3

F/C. Calculations reveal that the required memory starts exceeding the last level cache size for inputs larger than $N = 2^5$, which explains the dramatic drop off in performance. The same phenomenon applies for `base_no_opt` and `base_stdc` but is less apparent.

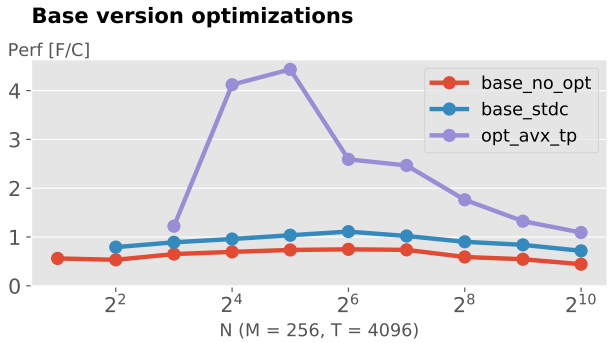

Fig. 8. Performance comparison of the base version optimizations. states.

Speedup N	`base_stdc`	`base_avx_tp`
`base_no_opt`	1.62	5.83
`base_stdc`	1	4.2

Table 2. Speedup comparisons of N-plots in base versions. An entry (*row*, *col*) denotes the speedup of implementation *col* compared to implementation *row*.

Memory version optimizations. The performance N-plots of the memory version optimizations can be seen in figure 9. Similarly to the base version plot, we observe a performance drop for all implementations as soon as the data exceeds the last level cache. In this series of optimizations, we treat `mem_stdc` as our baseline model and compare it to further optimizations applied to it. Transposing the input and output matrices results in significantly improved spatial locality, which is reflected in the performance of `mem_stdc_tp`: We observe a consistent speedup of up to 2.88 compared to `mem_stdc_tp`. Remarkably, `mem_stdc_tp` even outperforms `mem_avx` which is the direct vectorization of `mem_stdc`. The best performance is achieved by `mem_avx_tp`, which combines both techniques: The input is transposed, and AVX2 intrinsics are used. We observe a performance peaking at 3.45 F/C which is 21 % of peak performance and a speedup of up to 4.25 compared to `mem_stdc`.

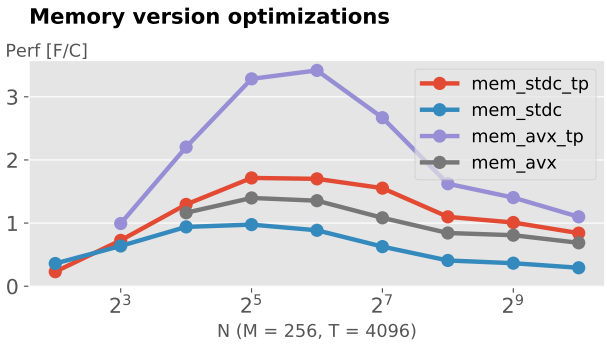

Fig. 9. Performance comparison of the memory version optimizations.

Table 3 compares the performance speedups between the different optimizations.

Speedup N	mem_stdc_tp	mem_avx	mem_avx_tp
mem_stdc	2.88	2.35	4.25
mem_stdc_tp	1	0.9	2.01
mem_avx	1.43	1	2.52

Table 3. Speedup comparisons of N-plots in memory versions. An entry (*row*, *col*) denotes the speedup of implementation *col* compared to implementation *row*.

Runtime comparison of both versions. Since base and memory versions each execute a significantly different amount of flops, comparing their performance is not a suitable measure. Therefore, we need to compare their runtime to see which one performs better.

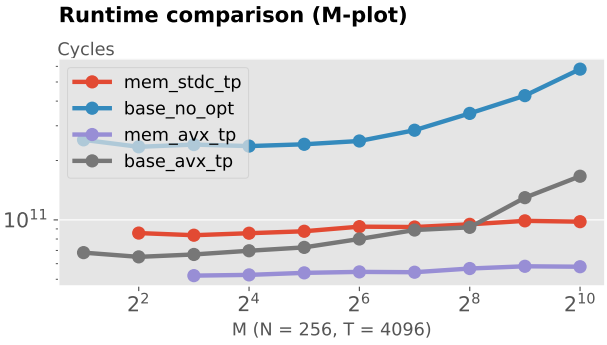

Fig. 10. Runtime comparison of the base version and memory version (both axes are in log-scale).

The advantage of the memory approach can best be seen when fixing N and T and varying M as figure 10 shows. Because the memory version has a better time complexity of $\mathcal{O}(N^2T + NM)$, compared to the original $\mathcal{O}(N^2T +$

NMT), the runtime is dominated by the N^2T member until M becomes sufficiently large. That is why even the standard C memory version for N=256, T=4096 and M=512 and 1024 has better runtime than `base_avx_tp`, which is the base version optimized using AVX2.

Additionally, even when varying N, the memory version is superior. Figure 11 shows that. `mem_avx_tp`, the best memory version optimized with AVX2, it takes only 64 % of the time compared to `base_avx_tp`, and around 20 % of the baseline version.

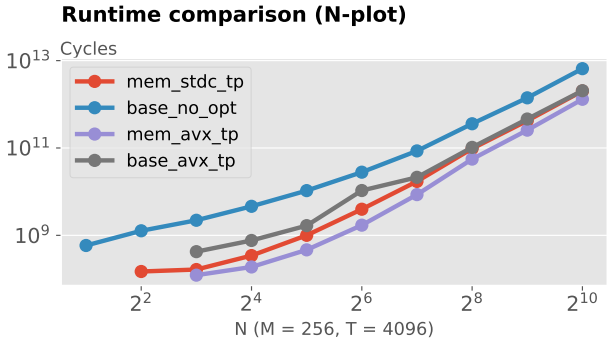

Fig. 11. Runtime comparison of the base version and memory version (both axes are in log-scale).

5. CONCLUSIONS

In this project, we have taken a straightforward C implementation of Baum-Welch and profiled it for optimization potential. We successively applied transformations to our code, where the profile suggested it to be beneficial. We transformed the implementation using well-known C techniques in support of the compilers' optimization capabilities, improvements to temporal and spatial memory locality and, finally, vectorization using Intel's AVX2 intrinsics framework.

Additionally, we noticed our implementation to be strongly compute-bound and decided to store additional values, leveraging a trade-off between memory and computational effort in hopes of an additional speedup. We found our optimizations to increase performance by a factor of up to 5.83 in the base version and 4.25 in the memory heavy variant, thus achieving a maximum 27 % and 21 % of theoretical peak performance, respectively. Additionally, the best memory version saves up to 80 % of the computation time as compared to the baseline.

These results clearly show the potential of optimizing Baum-Welch along a system-targeted dimension. As mentioned, previous work has focused primarily on algorithmic improvements and the creation of parallel versions of Baum-Welch. We are convinced that some of our results can translate to these previous findings, for example, applied to clusters with homogeneous compute nodes. Thus, we feel the next step for obtaining fast Baum-Welch implementations is to assess the applicability of our methods to parallel code and algorithmically improved versions, such as those employing fast state discovery.

6. REFERENCES

[1] Lawrence Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," in *Proceedings of the IEEE*, 1989.

[2] James K. Baker, "The dragon system—an overview," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 1975.

[3] Charles Wright, Lucas Ballard Scott Coull, Fabian Monrose, and Gerald Masson, "Spot me if you can: Uncovering spoken phrases in encrypted voip conversations," *IEEE International Symposium on Security and Privacy*, 2008.

[4] Chris Burgeand and Samuel Karlin, "Prediction of complete gene structures in human genomic dna," *Journal of Molecular Biology*, 1997.

[5] Tingting Liu and Jan Lemeire, "Efficient and effective learning of hmms based on identification of hidden states," *Mathematical Problems in Engineering*, 2017.

[6] Sajid Siddiqi, Geoffrey Gordon, and Andrew Moore, "Fast state discovery for hmm model selection and learning," in *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AI-STATS)*, 2007.

[7] Istvan Miklos and Irmtraud M. Meyer, "A linear memory algorithm for baum-welch training," *BMC Bioinformatics*, 2005.

[8] Imad Sassi, Samir Anter, and Abdelkrim Bekkhoucha, "A new improved baum-welch algorithm for unsupervised learning for continuous-time hmm using spark," *International Journal of Intelligent Engineering & Systems*, 2019.

[9] Dawei Shen, "Some mathematics for hmm," 2008.

[10] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Number 248966-043. May 2020.

[11] Jagode H. You H. Dongarra J. Terpstra, D., "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing*, ZIH, Dresden, 2010, pp. 157–173, Springer Berlin Heidelberg.