VITERBI ALGORITHMS FOR SPEECH RECOGNITION

Alessandro Dovis, Matteo Panzacchi, Martynas Pumputis

How to Write Fast Numerical Code, Spring 2014
ETH Zürich
Zürich, Switzerland

ABSTRACT

The Viterbi algorithm is very important in signal analysis and processing, but too little effort has been made by the research community to optimize its general case. In this paper we describe the optimizations we applied; we were able to achieve a performance boost of 15x, after exploiting spatial locality, compiler flags and SIMD vectorization. We prove that we are memory bound in the final implementation and describe some final changes in the algorithm that can lead to further improvements.

1. INTRODUCTION

The Viterbi algorithm is ubiquitous in computer science and electrical engineering today; it is widely used in digital telecommunication, speech recognition and synthesis, natural language processing and many other fields. Although some work has been done to create fast implementations of the Viterbi algorithm in some specific domains (e.g. convolutional codes), we could not find any example work that tackles the general case of Hidden Markov Models (HMMs). We bring speech recognition as an application of Hidden Markov Models and try to improve its performance by exploiting optimization techniques learnt in the course.

Motivation. Speech recognition, as explained in [1], can be viewed algorithmically as a pipeline: an initial feature extraction phase is followed in series by a phonetic, word and task recognition block, that respectively infer phonemes, words and sentences. In all these cases a sequence of observations (or symbols) is given as input and the block is supposed to output a sequence of states that with maximum likelihood matches the observations. More specifically:

- the phonetic recognition block uses the acoustic model to produce the most likely phoneme corresponding to a sequence of sounds (feature vectors);
- the word recognition block uses the lexical model to produce the most likely word corresponding to a sequence of phonemes;
- the task recognition block uses the grammar model to produce the most likely sentence corresponding to a sequence of words.

The problem of finding the most likely sequence of states, given an input observation sequence and a Hidden Markov Model (that we will describe below in further detail) is fundamental in speech recognition and this use-case is the focus of our work. Our implementation is general enough to be applied also in other applications that exploit the Viterbi algorithm to solve general HMMs.

Our implementation starts from the base pseudocode in [2] and exploits data structure optimizations, code restructuring, compiler optimizations and SIMD vectorization, to achieve a 15x performance increase with respect to the initial code.

Related work. Concerning domain specific implementations, we refer to [3] and [4] as important works in the field of convolutional codes. In particular they exploit the domain specific structure of the problem (in terms of ”butterflies”) to perform memory hierarchy optimizations, operation restructuring and vectorization.

In the speech recognition domain - and in few other applications - no specific reduction of the problem can be performed, so the general HMM has to be employed. Most scientific environments have an implementation of HMM basic functionalities (defining and generating a model, decoding an input sequence, etc.), but their performance is very poor, as shown in a later section.

2. BACKGROUND: HMM AND VITERBI ALGORITHM

A Hidden Markov Model explains a Markov process with hidden states, whose manifestation is a sequence of observations. Formally it comprises:

- an observation space (set of observations/symbols) of cardinality E
- a state space (set of states) of cardinality S

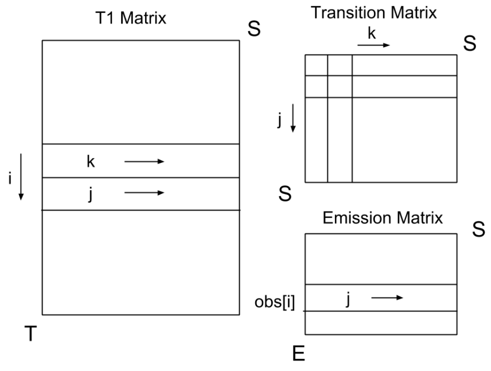

Fig. 1. Data structure access pattern in Viterbi algorithm

- a transition matrix (SxS), whose entry TM_{ij} is the probability of transitioning from state i to state j
- an emission matrix (SxE), whose entry EM_{ij} is the probability that state i is observed as symbol j

Given a sequence of observations *obs* of length T, the Viterbi algorithm is used to provide the most likely sequence of states that has generated it.

Algorithm and Data Structures. The Viterbi algorithm uses a dynamic programming approach to solve the problem. From now on, we will consider a transposed version of TM and EM (the transition and emission matrices), since they make the explanation of data accesses easier. Let us call P_i the prefix of the output state sequence of length i. A dynamic programming matrix T1 (TxS) will contain in position $T1_{ij}$ the probability that P_i ends with state j.

The algorithm computes iteratively row i in terms of row (i-1) using the recursive formula (expressed in C-like notation):

$$T1[i][j] = \max_k\{T1[i-1][k] \cdot EM[obs[i]][j] \cdot TM[j][k]\}$$

where $obs[i]$ is the i-th symbol of the observation sequence and the indices are referring to the transposed matrices and are exactly mapped to the triple nested ijk loop used in the code. The $argmax$ is kept in a secondary matrix T2 (TxS), which is then used to retrieve the sequence of states by traversing it backward.

The data access of the main matrices is explained graphically in Fig. 1. As we can see, each row of T1 is causally dependent on the previous row; furthermore, in each of the T-1 steps of the i-loop (row-by-row in T1), TM is fully scanned once without any reuse; only one row of EM is used in each step of the i-loop. These observations imply that no blocking can be applied to either matrix.

Cost Analysis. The cost analysis of the algorithm is straight-forward. The ijk loop seen before gives us an asymptotic cost in $O(T \cdot S^2)$.

It is possible to define a cost measure as:

$$C(S,T) = (fl.mults, fl.divs, fl.logic)$$

The resulting cost of the algorithm is:

$$(S + 2S^2(T-1), S(T-1), S + S(T-1) + S^2(T-1))$$

where divisions come from normalization steps (needed for correctness), while logic operations are the max operations (comparable to additions).

3. OPTIMIZATIONS

In the following section we discuss optimizations applied to the baseline implementation of Viterbi algorithm.

Baseline. The initial version of Viterbi algorithm implementation is based on pseudo-code found in [2] and was implemented by the authors, because we could not find any generic Viterbi algorithm code.

The algorithm implementation uses matrices as a main data structure. Each matrix is represented as an array of references pointing to a list of elements which might be either a probability stored as *double* or a state identifier stored as *int*. Also, each row is allocated via *GNU libc malloc(3)* function call.

The transition matrix and *T1* matrix are accessed column-wise inside innermost *ijk* loop. The same access pattern is held for the emission matrix.

After calculating the likelihood (probability) of each state for the *i-th* observed symbol, we do a normalization step to avoid correctness problem that occurs when multiplying small probabilities whose product becomes equal to zero.

Data Structure Optimizations. The matrix allocation strategy used in the baseline implementation is not optimal in terms of spatial locality and also TLB cache accesses, because each row might reside in different memory pools used by *malloc(3)* ([5]). Also, this approach wastes memory due to a header needed for *malloc(3)*. The better way would be to allocate a chunk of memory capable to store the whole matrix and divide it into rows, i.e. linearize the matrices into 1-D arrays.

Next, the column-wise access pattern of matrices used in the baseline implementation has poor spatial locality for large matrices. As an improvement, we transposed the emission and the transition matrices before using them in the algorithm. This modification changed the access pattern to row-wise and therefore it improved the spatial locality.

Code Restructuring. After identifying the most costly operation which is the division used in the normalization step, we replaced it with multiplication by the multiplicative inverse of the *max*. This optimization reduced the number of divisions from $S \cdot (T-1)$ to $T-1$.

As a next step, we moved common subexpressions outside a loop when possible. It includes reading values from the observation sequence array and the emission matrix. This optimization reduces code size and useless accesses to memory, which might result in less instruction and data cache misses.

Compiler Optimizations. When normalizing probabilities in *T1* matrix or finding the best state for the i-th observation inside the *ijk* loop, we introduce a new branch used if a condition holds. In both cases we look for the maximal probability in an array, therefore the condition consists of comparing a value in the array with a current maximum. It is known that the average number of assignments needed to find the maximum value is the n-th Harmonic number ([6]) which means that the condition is very unlikely in most cases. Therefore we can provide *unlikely* hint for the compiler to remove the branch from common path which reduces the code size of the path.

Nowadays, compilers are able to do many optimizations. For some of them special flags are needed. We compiled our optimized code with *gcc's -ffast-math and -funroll-loops* optimization options, although we tried many others. Also, different compilers might produce machine code with different performance. For this reason, we tried *gcc*, *icc* and *Clang* compilers and we picked the one with best performance (trying several flags in all of them).

SIMD Optimizations. Although after doing memory analysis it was clear that the algorithm is memory-bound for large matrices - see below -, we did k-loop unrolling. Later, this allowed us to use *AVX* instructions to parallelize the operations needed in the unrolled version for the search of a maximal value. The vectorized version of the search is depicted in Fig. 2. As seen in the analysis, this gave us a speed-up, because of a better usage of the available bandwidth.

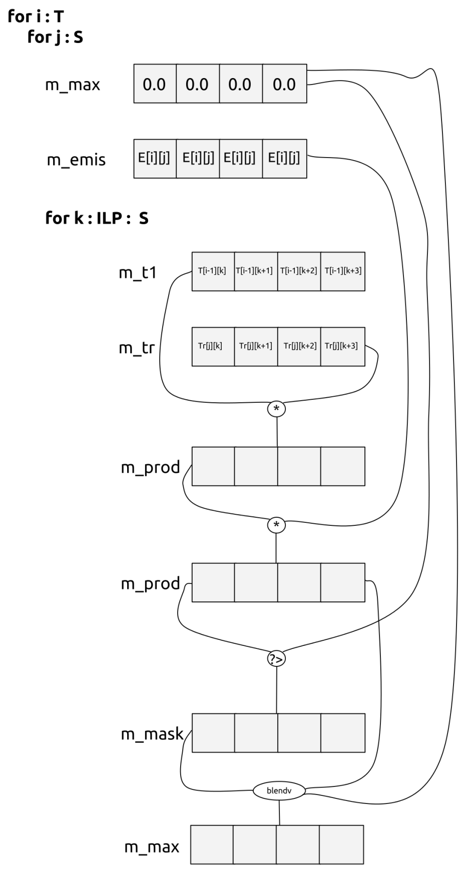

Fig. 2. SIMD implementation scheme

4. EXPERIMENTAL RESULTS AND ANALYSIS

In this section we are going to present the results we got for the different optimizations. In the final subsections we will also explain the memory boundness analysis that we did. The reasons of this analysis will be clear after seeing the performance plot of the non vectorized code.
In the beginning, we tried to look for an already existing benchmark code that we could use as comparison for our best code. Unfortunately, all the implementations that we found were not suited for our use case (i.e. speech recognition). Next, we tried to use as benchmark code the corresponding Viterbi implementations of Matlab and R. However, as shown in Fig. 3, our base implementation (with no optimizations) is already faster than the one in Matlab and R. Consequently, we decided not to use any benchmark code as comparison for our solution.

Experimental setup. We did all of the experiments on a dual core Ivy Bridge machine with a clock frequency equal to 2.3GHz. In case of balanced number of additions and multiplications, the scalar peak performance of this machine is 2 flops/cycle (1 add/cycle and 1 mult/cycle). However, in our algorithm we have a ratio of 2:1 of mults and logic ops (which can be seen as additions), as a result the peak performance is 1.5 flops/cycle instead of 2. In the case of vectorized code, since the machine is provided with AVX, it has a theoretical peak performance equal to 8 flops/cycle, but we still have and unbalanced ratio in the operations so a tighter peak performance would be 6 flops/cycle.
For the first range of experiments (all the optimizations apart from SIMD), the code was compiled with *gcc* and the following flags:

```
-O3 -fno-vectorize
```

In addition, we have used architectural flags specific for Ivy Bridge. Then, as mentioned in section 3, in the last 2 scalar optimizations we tried out different compiler flags.
We have also performed the same experiments with other two compilers (*icc* and *clang*), but - as shown later - we had the best performance with *gcc*.
Firstly, we ran some benchmarks by varying one of the three different parameters (#states, #symbols and #observations)

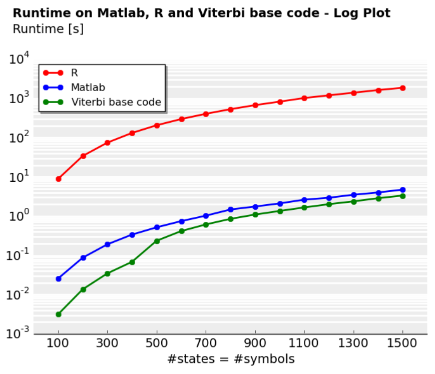

Fig. 3. Runtime of our base implementation compared to Matlab and R implementations of the Viterbi algorithm (log plot)

at a time. From these initial benchmarks we could infer that the only parameter which really affects the performance is the number of states. As a consequence, we decided to perform all the following experiments using squared matrices (i.e. #states=#symbols). The range of values that we used was going from 100 to 1500 with an increment of 100 every time. Furthermore, we executed stress tests for sizes up to 15000. However, since the performance trend does not change for bigger values, we will omit these results from here on.

Optimization Results. As shown in Fig. 4, we had the main performance boost when we transposed the matrices, because we achieved a better spatial locality, since we started accessing the matrices row-wise instead of column-wise. Indeed, by only transposing the matrices we moved from 0.1 flops/cycle to 0.8 flops/cycle with a speed-up of 8x. With the other optimizations we got a very little speed-up, reaching 0.9 flops/cycle with compiler optimization flags, which brings a 9x speed-up from the starting code.

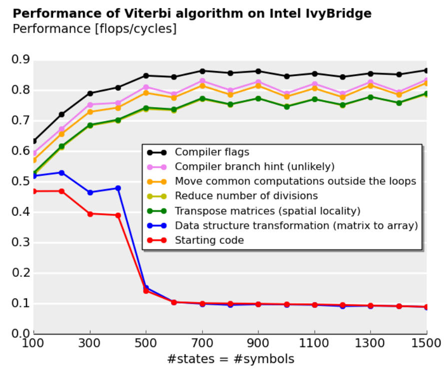

Fig. 4. Plot of the different optimized versions of the code without considering SIMD

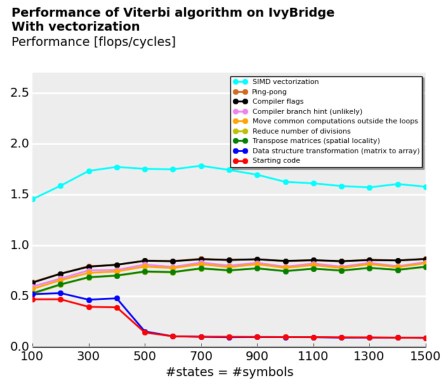

Fig. 5. Plot of all the different optimized versions of the code

One thing that it is worth mentioning is that the performance for the optimized code stays stable until 1500. As a consequence, we tried to stress test the code going up to matrices of size 15000x15000 in order to check if the performance would drop or not. As mentioned before, even with very high numbers the performance did not drop, and these results led us to assume that we were memory bound. In order to check whether or not our assumption was right we performed the roofline plot and the memory boundness analysis described in the following paragraphs.
In Fig. 5 the plot with all the optimizations is shown. As can be seen we reached approximately 1.5 flops/cycle by passing to vectorized code and we got a significant speed-up of 15x from the starting code.

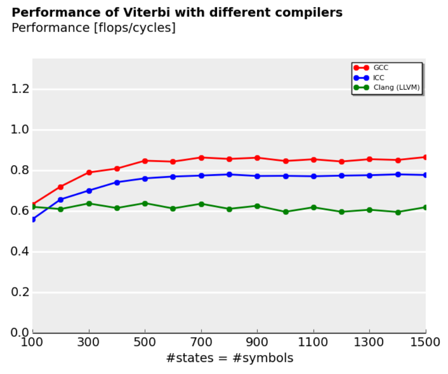

Fig. 6. Comparison of different compilers

Finally, we tried to compile the best not vectorized code with different compilers trying out compiler-specific flags and - as shown in Fig. 6 - we had the best performance with *gcc*.

Operational intensity. To get more confident about our memory boundness assumption, we firstly decided to calcu-

late the upper bounds of the Operational Intensity.
A loose upper bound is given when all the data we are accessing fit into cache. As discussed before, we have the following asymptotic number of floating point operations:

$$W(S, E, T) = O(T \cdot S^2)$$

Then, since the data fit into cache we would load from memory every element of the matrices that we use only once. Consequently, the number of bytes loaded from memory is equal to:

$$Q(S, E, T) = O(S^2 + S \cdot E + 2 \cdot S \cdot T)$$

which can be simplified into $O(S^2)$, since we are considering square emission and transition matrices ($S = E$) and T is always smaller than S. Thus, $I(S, E, T) = O(T)$. For a tighter bound that applies to bigger matrices, we assume that the working set does not fit into cache. As a consequence, we are loading from memory every time we access an element that was previously evicted. The number of floating point operations does not change, but the number of bytes loaded from memory becomes:

$$Q(S, E, T) = O(T(S^2 + S + 2 \cdot S)) = O(T \cdot S^2)$$

The number of accesses to the transition matrix changes because we are loading the whole matrix at every iteration (supposing an LRU replacement policy in caches). Consequently, we have T loads of the matrix. In this case the Operational intensity is $I(S, E, T) = O(1)$. This is a good indication that we are probably memory bound for big sizes.

Bandwidth analysis. In this paragraph we explain the analysis that we did in order to check if we were using the whole memory bandwidth.
Firstly, we measured the bandwidth with [7] and [8], and with both of them the bandwidth was roughly 4.4 B/cycle. We used this result in our back-of-envelope analysis.
For every experiment that we did on the various sizes we took the runtime of the best scalar code and the SIMD code. Then we have calculated the number of bytes that we could transfer during the whole execution with the following formula:

$$\#max\_trasferred\_bytes = Bandwidth * \#cycles$$

Finally, we compared this result with the approximate number of bytes effectively loaded from memory, and since for big sizes the data does not fit in cache, it is well approximated by the number of loads of the whole transition matrix:

$$\#approx\_transferred\_bytes = T \cdot S^2 \cdot 8$$

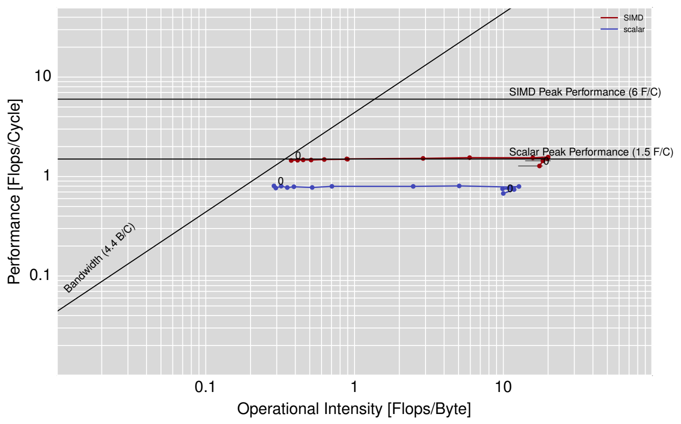

Fig. 7. Roofline plot of scalar and vectorized versions of Viterbi algortihm

Computing the ratio between these two quantity we could infer the effective bandwidth that was used during the execution. The results show that with the scalar code we were using half of the available bandwidth (i.e. *ratio=0.5*), whereas with the vectorized code we were using the whole bandwidth.
These results gave us higher confidence that we were memory bound for large sizes of the matrices. The increase in performance with the introduction of vectorization can be explained by a better usage of the available bandwidth.
A final proof of memory boundedness is given in the following paragraph with the roofline model.

Roofline plots. As final step of the memory boundness analysis we did the roofline plot with [7]. As shown in Fig. 7 for both scalar and vectorized versions of Viterbi we are compute bound for small sizes (data points on the right side). As we increase the size, the lines start to move towards the memory bound part, until for very large size we are in both versions memory bound (data points on the left side). However, for the scalar version we do not hit the memory bound line, because we do not use the whole bandwidth, whereas in the vectorized code we do. This is a further confirmation of our initial assumption and of the analysis that we explained in the previous paragraph.

5. FURTHER IMPROVEMENTS AND FUTURE WORK

Log-Domain and Single Precision Floats.

We have focused so far on the algorithm in the natural domain; the algorithm can be easily transformed into log domain, though: if an element-wise logarithm is performed on both TM and EM and the multiplications are substituted by additions, the algorithm still works. This is formally guaranteed by the monotonicity of the logarithm

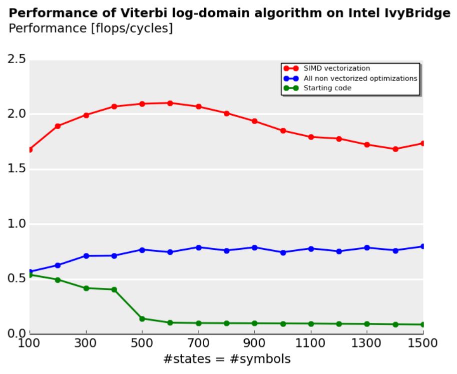

Fig. 8. Viterbi algorithm optimizations in log domain

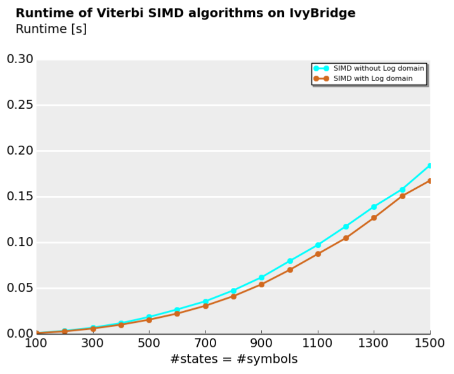

Fig. 9. Runtime comparison between vectorized code in natural and log domain

function (that leaves the correctness of the max operation untouched).

In this case, the multiplications are transformed into additions, so we would have a different cost measure:

$$C(S,T) = (fl.adds, fl.logic) = \\ (S + 2S^2(T-1), S + S^2(T-1))$$

and the peak performance in the scalar case would be 1 (because comparisons are equivalent to additions).

In Fig. 8 we see that the qualitative behaviour of the performance in both scalar and vectorized case is the same as in natural domain. The runtime is although lower (see Fig. 9).

Next, we tried to explore the impact of reducing the size of the data structures by replacing double precision floating point numbers with single precision; the approximation of the result given by this solution is too inaccurate to pass our validation, therefore we omit further details about it.

An interesting future experiment would be to deploy our optimized code onto an Atom microprocessor - where the cost of addition and multiplication is very unbalanced - and see if the runtime gain is more evident.

6. CONCLUSIONS

We implemented the Viterbi algorithm in its general form, focusing on its application in speech recognition. As a result of the optimizations described in the paper, we were able to achieve a 15x performance gain.

We proved that we hit the memory bound in the final implementation, by analyzing the operational intensity, bandwidth usage and roofline plot.

Finally, we translated the algorithm to log-domain, analyzed it in this new shape and proposed some further improvements and topics of investigation.

As far as we know, this is the first work that tries to optimize the Viterbi algorithm for such a general scope; many techniques we have exploited could be used in any dynamic programming algorithm with a similar structure.

7. REFERENCES

[1] R. Giancarlo A. L. Buchsbaum, “Algorithmic aspects in speech recognition: An introduction,” *JEA*, vol. Volume 2.

[2] Wikipedia, “Viterbi algorithm,” http://en.wikipedia.org/wiki/Viterbi_algorithm.

[3] P. Karn, “Fec library version 3.0.1,” http://www.ka9q.net/code/fec/, August 2007.

[4] F. de Mesmay et al., “Computer generation of efficient software viterbi decoders,” in *Proc. International Conference on High Performance Embedded Architectures and Compilers (HiPEAC), Lecture Notes in Computer Science*, Springer, Ed., 2010, vol. Vol. 5952, pp. pp. 353–368.

[5] GNU Project, “The gnu c library,” http://www.gnu.org/software/libc/libc.html.

[6] Stack Overflow, “Number of assignments necessary to find the minimum value in an array?,” http://stackoverflow.com/questions/6735701/number-of-assignments-necessary-to-find-the-minimum-value-in-an-array.

[7] G. Ofenbeck, “Perfplot,” https://github.com/GeorgOfenbeck/perfplot.

[8] C. Staelin L. McVoy, “Lmbench - tools for performance analysis,” http://www.bitmover.com/lmbench/.