SINGLE CORE OPTIMIZATION FOR BAUM-WELCH

Lukas Käppeli, Steven Stalder, Andreas Roth, Kenan Bešić

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

The Baum-Welch algorithm is a widely used method to find the unknown parameters of hidden Markov models. We focus on optimizing this algorithm for single core execution and demonstrate how we move from a straightforward implementation to a highly optimized one. We describe how to enhance spatial and temporal locality of memory accesses and present optimizations improving computational parallelism. Our methods improve the performance by a factor of up to 51.8x compared to a third party implementation.

1. INTRODUCTION

The Baum-Welch algorithm is an instance of the expectation-maximization (EM) algorithm, a statistical procedure to determine maximum likelihood estimates of probabilistic models. Its purpose is to discover the unknown parameters of a hidden Markov model (HMM) in an iterative process, where a set of arbitrary initial probabilities is updated to better represent a sequence of observed states. A repeated execution of the algorithm on the same observation values will let the parameters of the HMM converge to a local maximum.

Motivation. HMMs are used in numerous areas such as cryptanalysis [1, 2, 3], bioinformatics [4, 5, 6] and speech recognition [7]. However, since input data can grow very large and time is valuable, the need for highly performant implementations of the training process is real. While there are various implementations of the Baum-Welch algorithm for different infrastructural settings [8, 9], the goal of this work is to provide a version that is highly optimized for a single core. To achieve this, we utilize numerous optimization techniques which compilers are not able to perform themselves or only in a suboptimal manner.

Contribution. In this paper, we present a highly optimized implementation of one iteration of the Baum-Welch algorithm using the AVX2 instruction set. We achieve a performance of up to 7.0 flops/cycle on an Intel Haswell processor and outperform ParaHMM [10] – an existing implementation also utilizing AVX2 vector instructions – by a factor of up to 51.8x in runtime. Compared to a naive implementation of the algorithm, we improve the performance by a factor of up to 136.8x and compared to a version with the exact same number of floating point operations we still improve by a factor of up to 41.1x. Although we consider only one iteration to evaluate the performance, our implementation can easily be applied repetitively such that it runs until a condition for convergence is met. In that case, the speedup could be observed even more distinctly.

Related Work. This work optimizes the implementation of the Baum-Welch algorithm for a single core using double-precision floating point numbers. ParaHMM [10] optimizes the implementation of the algorithm for multiple cores using single-precision floating point numbers. Moreover, they optimize using AVX2 instructions. Therefore, their implementation serves as a good benchmark if run on a single thread. Similarly, HMMlib [9] optimizes for multiple cores using single-precision or double-precision floating point numbers using SSE instructions. However, many machine learning libraries support executions on GPUs and cuHMM [11] shows the possibilities that optimizations on the Nvidia CUDA platform may bring.

Chapter 2 of this paper provides the reader with the necessary background for the underlying algorithm. Furthermore, it contains the cost analysis for floating point operations and data movement from and to main memory. In Chapter 3, we present our approach and the different optimization methods used. Finally, Chapter 4 contains our evaluation on the performance gains of the different techniques.

2. BACKGROUND

Mathematical Background. Let N be the number of states in a model and let M denote the number of distinct observations. Furthermore, consider a set of observations $O = o_0, o_1, \ldots, o_{T-1}$. A hidden Markov model $\lambda = (A, B, \pi)$ consists of a state transition probability matrix $A \in N \times N$, an observation symbol probability matrix $B \in N \times M$ and an initial state distribution vector π of length N. In order to train such a model, i.e. maximizing $Pr[O|\lambda]$ for a given O, the Baum-Welch algorithm can be applied. The algorithm consists of three steps: computing the forward variable α,

the backward variable β and updating the model λ. Note that $\forall t < T : o_t \in [0, M-1]$, which means o_t can be used as an index for the matrix B. The values of α and β are in $\mathcal{O}(1/N^2)$ and therefore converge to zero very fast as N increases. To provide numerical stability we implemented the scaling trick explained in [12] and define α and β using scaling factors $c_0, c_1, \ldots, c_{T-1}$.

For $\alpha \in T \times N$, $i \in [0, N-1]$ and $t \in [0, T-2]$:

$$\alpha_{0i} = c_0 \pi_i b_{io_0},$$
$$\text{where } c_0 = \frac{1}{\sum_{i=0}^{N-1} \pi_i b_{io_0}}$$
$$\alpha_{t+1i} = c_{t+1}\Big(\sum_{j=0}^{N-1} \alpha_{tj} a_{ji}\Big) b_{io_{t+1}},$$
$$\text{where } c_{t+1} = \frac{1}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} \alpha_{tj} a_{ji} b_{io_{t+1}}}$$

For $\beta \in T \times N$, $i \in [0, N-1]$ and $t \in [0, T-2]$:

$$\beta_{T-1i} = c_{T-1}$$
$$\beta_{ti} = c_t \sum_{j=0}^{N-1} a_{ij} b_{jo_{t+1}} \beta_{t+1j}$$

To update A, B and π, we use the following equations, where $i, j \in [0, N-1]$ and $k \in [0, M-1]$:

$$\bar{\pi}_i = \frac{\alpha_{0i}\beta_{0i}}{c_0}$$
$$\bar{a}_{ij} = \frac{\sum_{t=0}^{T-2} \alpha_{ti} a_{ij} b_{jo_{t+1}} \beta_{t+1j}}{\sum_{t=0}^{T-2} \alpha_{ti}\beta_{ti}/c_t}$$
$$\bar{b}_{jk} = \frac{\sum_{t=0,k=o_t}^{T-1} \alpha_{tj}\beta_{tj}/c_t}{\sum_{t=0}^{T-1} \alpha_{tj}\beta_{tj}/c_t}$$

Verification of the Implementation. To verify the correctness of the base implementation, we generated a testset using RHMM [13], a HMM library for the R programming language. We verified that our base implementation produces the same HMM as the RHMM library on randomly generated input data after one iteration of the algorithm. However, since the RHMM library does not calculate the updated initial probabilities, we have to prove the correctness for π differently. We can show that if B is updated correctly, then π will also be updated correctly. The update of B depends on the result of $\sum_{t=0}^{T-1} \alpha_{ti}\beta_{ti}/c_t$ for all i, which contains $\pi_i = \alpha_{0i}\beta_{0i}/c_0$. Since our update of B is verified with the RHMM library, the update of π thus has to be correct as well.

Having validated our base implementation, we compare its results to all subsequent optimized versions to confirm their correctness, too.

Cost Analysis. In this project, we measure performance in flops/cycle. For our optimized versions, we perform a total of $6N^2T - 4N^2 + 9NT + NM - 2N + 2T$ floating point operations. Furthermore, we are accessing $48N^2T - 16N^2 + 120NT + 24NM + 24N + 40T - 12$ bytes of memory during the computation. However, most of those memory accesses happen within the caches for reasonably small values of N, M and T. The only traffic between main memory and the caches happens for compulsory misses of additionally allocated arrays, which only amounts to $8NT + 8N + 16T$ bytes in total. Note that for the input matrices, we assume a warm cache scenario, which is reasonable as the Baum-Welch algorithm is an iterative procedure where the results of the previous iteration will be reused. This makes roofline analysis difficult since the operational intensity will always be very high (see Figure 1), while the bottleneck clearly lies in the data movement between caches. We therefore cannot determine a reliable memory-induced performance bound analytically. This problems remains also for very large inputs, i.e. when data will begin to get evicted from the L3 cache and written back to main memory due to limited cache capacity.

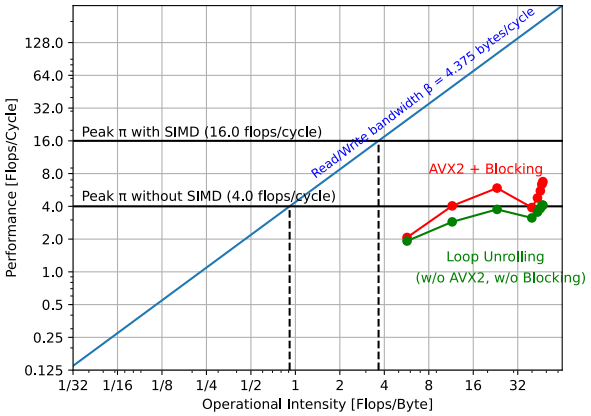


Fig. 1: Roofline model for some computations with parameter values N, M and T between 8 and 64. Memory bandwidth β measured with STREAM [14, 15].

3. OPTIMIZATIONS

In this section, we present our optimizations on the straightforward implementation using the formulas from Section 2. Our computations run on input matrices storing variables of type *double* for increased precision. We further assume

our input dimensions N, M and T are divisible by 8. We optimize our code for a Haswell processor which allows us to use AVX and AVX2 vector instructions.

Base Implementation. Our function takes the input and output variables as function arguments where the matrices are given in row-major format. This version is completely based on the mathematical background presented in Section 2 and serves as a naive non-optimized baseline. Besides the standard input arguments for one step of the Baum-Welch algorithm, we also require the caller to provide the matrices α and β.

Flop Reduction. The straightforward implementation contains many redundant floating-point operations. Therefore, our first optimization aims at removing those. In order to prevent unnecessary computations, we created three additional arrays which store intermediate results. This reduced the number of executed floating point operations without increasing the number of memory accesses. Note that compilers (e.g. g++) are capable of performing such optimizations to a certain degree by replacing common subexpressions. However, as those common subexpressions in our setting are in very different code locations, compilers are unable to perform those optimizations by themselves.
Finally, we also added a few operations. By replacing divisions of loop invariant variables by the multiplication of their inverse value, we actually perform an additional floating point operation, as shown in the following code snippet. However, reducing the number of divisions yields a much better performance due to their high latency and low throughput.

1: $c[0] \leftarrow 1/c[0]$
2: **for all** i in 0 to N **do**
3: $\quad alpha[i] \leftarrow alpha[i] * c[0]$
4: **end for**

Column Major. The nature of the algorithm implies better spatial locality when the matrices are stored in a column major format. This becomes obvious when considering the update function of the matrices A and B. By reordering some loops and switching to the column major format, we were able to access all two-dimensional arrays in a row-wise manner. This leads to a far lower amount of cache misses. In fact, for a cache with 64 Byte cache lines, one could reduce cache misses for doubles by up to a factor of 8. Due to the row-wise access, all elements which are loaded into the cache are more likely to be used before they are evicted from the cache.

Loop Unrolling. Unrolling loops by a factor of 8 and using multiple variables for intermediate computations of independent loop iterations supports the compiler's optimizations and achieves higher instruction-level parallelism. Interestingly, not all loops can be unrolled. The nature of the Baum-Welch algorithm is iterative over the temporal component. This means that for some loops the computations of an iteration depend on the results of the previous one. However, since only the outermost loops are dependent on the temporal component, the inability to unroll them is not a bottleneck.

Vectorization. By also applying AVX2 vector instructions, we were able to further improve the performance of the algorithm. However, two additional bottlenecks arose during the implementation stage of this optimization. First, in multiple segments of the forward procedure we are required to sequentially sum up values in a horizontal way, involving high-latency instructions such as `_mm256_hadd_pd` and `_mm256_permute2f128_pd`. The second bottleneck that becomes evident when applying vectorization is the number of memory accesses. A common pattern in the algorithm is shown in the following snippet:

1: load values from memory
2: perform a single operation on these values
3: write the values back to memory

While we successfully vectorized these code patterns, performance for higher input dimensions in these sections is defined by the data transfer from and to memory.

Using AVX2 vector instructions also allowed us to effectively enforce the usage of FMA units for specific code segments. Listings 1 and 2 show a code pattern we encountered in the forward procedure of our algorithm. Precomputing `beta * ct` in Listing 1 and adding the result to the two other variables are three floating point operations that can be performed in 11 cycles. However, using FMA units we are able to perform four floating point operations in 5 cycles (Listing 2). Although we perform one additional floating point operation, computations can be performed in fewer cycles.

Listing 1: A code pattern that requires 3 flops can be performed in 11 cycles

```
double temp = beta[t*N+i] * ct;
b_[index*N+i] += temp;
gamma[i] += temp;
```

Listing 2: The same code pattern as in Listing 1 that requires 4 flops and can be performed in 5 cycles using FMA instructions

```
b_[index*N+i] += beta[t*N+i] * ct;
gamma[i] += beta[t*N+i] * ct;
```

Blocking. Finally, we still encountered some suboptimal access patterns in some loops. More precisely, cached memory locations could get evicted from cache due to limited cache capacity, even though they would be accessed again in later loop iterations. To mitigate this problem of poor temporal locality, we implemented loop blocking for the affected parts of the code. As a consequence, the values of memory locations that need to be read or written to several times are more likely to remain in the L1 cache for every usage as they are only accessed when the loop iterates through the respective block. Obviously, this reduces the overall amount of cache misses for an appropriately chosen block size. For this implementation, we always used 4x4 or 8x8 blocks as they provide the best performance gains without limiting the input sizes too much.

Register Management. In our code, we always load and store the values from AVX registers explicitly. Thus, a lot of the code has a load-flop-store pattern. Since we have a lot of memory movements limiting the performance of the algorithm, we investigated further optimizations to reduce these movements. The idea was to let the compiler keep track of the available registers and let it manage the load and store operations to make them more efficient. Therefore we replaced the explicit load and store instructions by keeping a reference to each vector variable in memory. It is obvious that a __m256d variable needs 32 bytes of memory, since it contains 4 doubles. But the g++ compiler was not able to generate more efficient code, which is the reason why we did not implement this idea. By looking into the assembly code, one could see that performance loss is caused by many additional index computations. These are mainly created by the compiler, thus the explicit load and store instructions are more efficient than the compiler registry management.

4. EXPERIMENTAL RESULTS

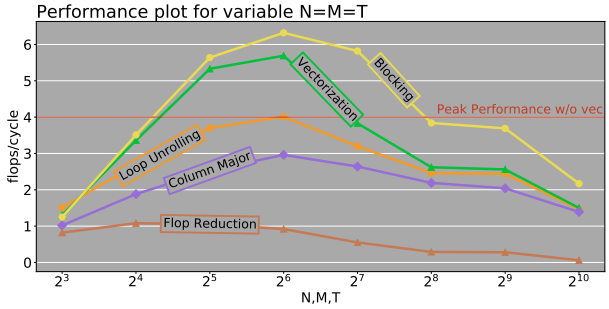
In this section, we present the results of our evaluations of the different optimizations that we applied to the initial naive base implementation. We first focus on the performance improvements gained by applying the optimization techniques from Section 3. Then, we present the runtime reductions compared to the naive base implementation and the benchmark. Note that all optimizations are applied on top of each other in the order they are discussed.

Experimental Setup. We ran the experiments on the following machine:

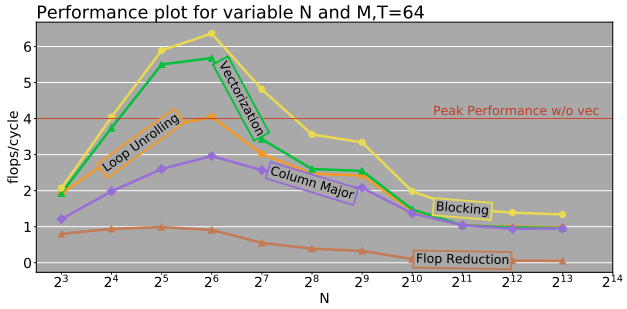
OS	*Ubuntu 20.04*
CPU	*Intel i7-4700MQ @ 2.4Ghz*
L1 Cache	*128 KiB*
L2 Cache	*1 MiB*
L3 Cache	*6 MiB*
RAM	*12GiB*

We compiled our implementations using *g++ (Ubuntu 9.3.0-10ubuntu2) 9.3.0* and used the following optimization flags: `-O3 -march=haswell -ffast-math`.

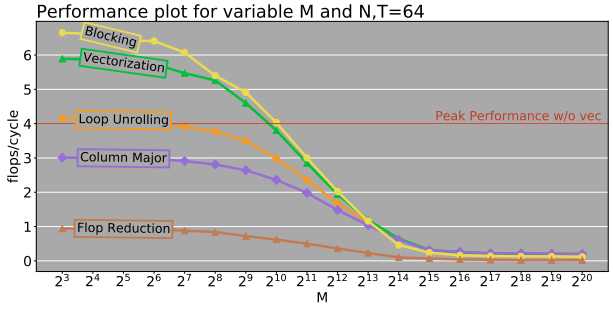
As explained in Section 2, the runtime of the Baum-Welch algorithm depends on three input variables: The number of distinct states N, the number of possible observations M and a time component T. Figure 2a shows the performance improvement when jointly increasing all three variables. For more detailed evaluations, we fix two of these variables to 64, while changing the value of the third variable. We only consider input variables in the range from 2^3 to 2^{20} that are divisible by 8.



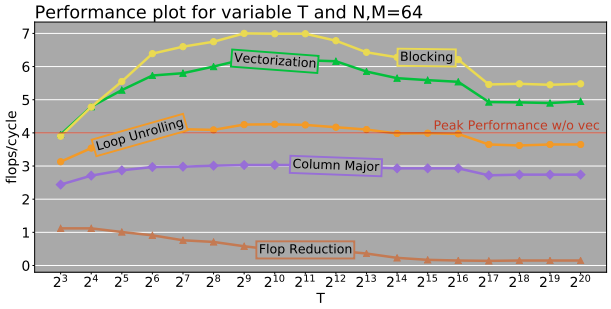
(a) Performance plot for N=M=T



(b) Performance plot for variable N



(c) Performance plot for variable M



(d) Performance plot for variable T

Fig. 2: Performance plots for joint increase of all three variables (a) and two fixed variables (b,c,d)

Flop Reduction. The Figures 3a, 3b and 3c show that *Flop Reduction* performs up to 26.8x better than the naive base implementation. Despite the enabled -O3 flag, there is a remarkable difference between the two versions. This means that we have implemented some optimizations which the compiler was not able to do itself. For large values of N, M or T, the improvement becomes negligible, indicating that there are other bottlenecks to consider.

Column Major. As the Figures 2a, 2b, 2c and 2d show, switching to column major format for the matrices removed one of the bottlenecks. Compared to *Flop Reduction*, we achieved a maximum speedup of 19.5x at $T = 2^{16}$ and $N, M = 64$. One reason for that is that the compiler could vectorize more loops due their row-wise access pattern on matrices. Furthermore, the lower number of cache misses results in less memory movement. We have measured that the optimized cache utilization and the ability of the compiler to vectorize loops both have an equal portion of the achieved speedup.

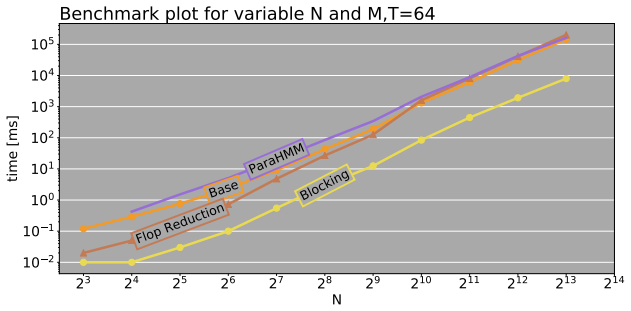
Loop Unrolling. In Figure 2b and Figure 2c we see that unrolling loops leads to a significant improvement in performance for smaller values of M or N. However, the larger the input sizes get, the less impact this optimization has on performance. Once the L3-cache size is exceeded, the performance becomes equal to *Column Major*. We see that memory accesses have a large impact on the performance. In contrast, Figure 2d shows that even for large values of T, *Loop Unrolling* achieves an improvement. This follows from the fact that the number of memory accesses scale quadratically in N but only linearly in T.

Vectorization. Implementing AVX2 vector instructions results in a performance improvement of up to 1.5x compared to *Loop Unrolling* (see Figure 2d) for higher values of T. For increased N and M there is almost no performance gain because the performance of the implementation is dominated by memory movements.

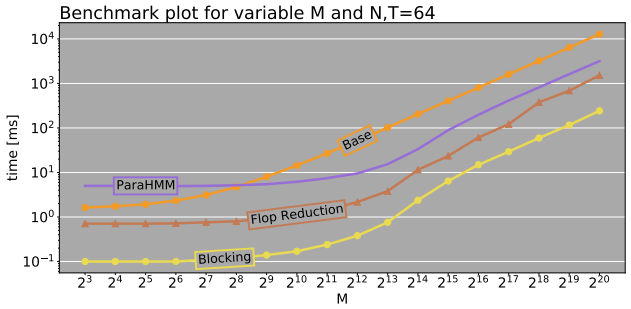
Blocking. Applying blocking on top of all the other optimizations resulted in the overall best performance for the algorithm. Compared to *Vectorization*, we achieve an improvement of up to 1.4x in performance for most parameter values due to the improved temporal locality in the caches.

The theoretical peak performance on the Haswell microarchitecture lies at 16.0 flops/cycle when using double-precision vector instructions. With our optimizations, we were able to achieve a performance of up to 7.0 flops/cycle, which corresponds to 44% of the theoretical maximum. As discussed at the end of Section 2, implementations of the algorithm are limited by memory movements between caches. Another bottleneck is the iterative nature of the algorithm which we documented in paragraph *Vectorization* of Section 3.

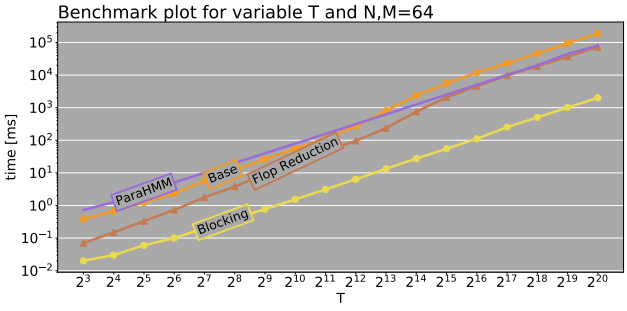
Benchmark. The following figures show our results compared to the benchmark ParaHMM [10], which is a Baum-Welch implementation using single-precision floating point numbers and optimized for multicore platforms. This causes a small overhead since we execute the benchmark on only one core. However, we outperform the benchmark by a very large factor of up to 51.8x in runtime which cannot be explained solely by this overhead.



(a) Benchmark plot for variable N



(b) Benchmark plot for variable M



(c) Benchmark plot for variable T

Fig. 3: Benchmark plots, fixing two out of the three variables

5. CONCLUSIONS

We presented an implementation of the Baum-Welch algorithm that outperforms an existing implementation in C by a factor of up to 51.8x. Starting from a naive base implementation, we reduced the number of floating point operations using memoization. Furthermore, we discussed how the storage format of matrices affects performance and spatial locality for large input sizes. For our implementation, we demonstrated the superiority in performance of the column-major matrix storage format. Moreover, we provided help to the compiler by unrolling loops such that instruction-level parallelism is increased. Introducing AVX2 vector instructions led to higher parallelism and enabled the enforcement of FMA instructions very easily. Finally, we further optimized memory accesses by implementing blocking in our nested loops which improved temporal locality of memory accesses and thus reduced the number of cache misses. The benefits of our various optimization efforts are very apparent when compared to our naive base implementation. Our fully optimized implementation achieves a peak performance of up to 7.0 flops/cycle. While we considered both computational as well as memory-related optimizations, we encountered and analyzed bottlenecks that are given by the nature of the algorithm and cannot be improved upon.

6. CONTRIBUTIONS OF TEAM MEMBERS

Lukas Käppeli. Started with Kenan on the base implementation. Mainly worked on converting the algorithm to use column major format and accessing arrays in a row wise manner. Unrolled some loops with Andreas and contributed to the implementation of blocking. Tried to reduce the memory movements as mentioned in Section 3, Register Management.

Steven Stalder. Worked on changing the algorithm to use the column major format. Vectorized a part of the code with AVX2 vector instructions. Implemented blocking for two of the loops. Also did some minor optimizations for reducing memory accesses and improving port utilization.

Andreas Roth. Focused on removing aliasing from the base implementation. Together with Lukas, I worked on unrolling some loops and implemented vectorization on top of some of the unrolled loops. After that, I worked on implementing blocking in the update step.

Kenan Bešić. Supported Lukas with the base implementation. Afterwards worked on reducing the flop count. Contributed to the unrolling of the loops and then worked on vectorizing the code.

7. REFERENCES

[1] Janis Dingel and Joachim Hagenauer, "Parameter estimation of a convolutional encoder from noisy observations," in *2007 IEEE International Symposium on Information Theory*. IEEE, 2007, pp. 1776–1780.

[2] Charles V Wright, Lucas Ballard, Scott E Coull, Fabian Monrose, and Gerald M Masson, "Spot me if you can: Uncovering spoken phrases in encrypted voip conversations," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 35–49.

[3] Billy Bob Brumley and Risto M Hakala, "Cache-timing template attacks," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009, pp. 667–684.

[4] Jan O Korbel, Alexander Eckehart Urban, Fabian Grubert, Jiang Du, Thomas E Royce, Peter Starr, Guoneng Zhong, Beverly S Emanuel, Sherman M Weissman, Michael Snyder, et al., "Systematic prediction and validation of breakpoints associated with copy-number variants in the human genome," *Proceedings of the National Academy of Sciences*, vol. 104, no. 24, pp. 10110–10115, 2007.

[5] Chris Burge and Samuel Karlin, "Prediction of complete gene structures in human genomic dna," *Journal of molecular biology*, vol. 268, no. 1, pp. 78–94, 1997.

[6] Arthur L Delcher, Kirsten A Bratke, Edwin C Powers, and Steven L Salzberg, "Identifying bacterial genes and endosymbiont dna with glimmer," *Bioinformatics*, vol. 23, no. 6, pp. 673–679, 2007.

[7] James Baker, "The dragon system–an overview," *IEEE Transactions on Acoustics, speech, and signal Processing*, vol. 23, no. 1, pp. 24–29, 1975.

[8] Holger Wunsch, *Der Baum-Welch Algorithmus für Hidden Markov Models, ein Spezialfall des EM-Algorithmus*, Ph.D. thesis, Master's thesis, Universität Tübingen, 2001.

[9] A. Sand, C. N. S. Pedersen, T. Mailund, and A. T. Brask, "Hmmlib: A c++ library for general hidden markov models exploiting modern cpus," in *2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology*, 2010, pp. 126–134.

[10] Yuchen Huo and Danhao Guo, "ParaHMM," Available at: `https://github.com/firebb/parahmm`.

[11] Chuan Hui Liu, "cuHMM : a CUDA Implementation of Hidden Markov Model Training and Classification," 2009.

[12] Dawei Shen, "Some mathematics for HMM," 2008, Available at: `https://courses.media.mit.edu/2010fall/mas622j/ProblemSets/ps4/tutorial.pdf`.

[13] Ollivier Taramasco and Sebastian Bauer, "HMM-Fit function in the RHmm package for R," 2013, Available at: `https://cran.r-project.org/src/contrib/Archive/RHmm/`.

[14] John D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[15] John D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," Tech. Rep., University of Virginia, Charlottesville, Virginia, 1991-2007, A continually updated technical report. http://www.cs.virginia.edu/stream/.