Two Independent and Highly Efficient Open Source TKF91 Implementations

Nikolai Baudis, Pierre Barbera, Sebastian Graf, Sarah Lutteropp, Daniel Opitz, Tomas Flouri, and Alexandros Stamatakis

Karlsruhe Institute of Technology, Institute of Theoretical Informatics, Kaiserstrasse. 12, 76131 Karlsruhe, Germany
{daniel.opitz,sebastian.graf,nikolai.baudis}@student.kit.edu
http://www.informatik.kit.edu/

Abstract. In the context of a master level programming practical at the computer science department of the Karlsruhe Institute of Technology, we developed and make available two independent and highly optimized open-source implementations for the pair-wise statistical alignment model, also known as TKF91, that was developed by Thorne, Kishino, and Felsenstein in 1991. This paper has two parts. In the educational part, we cover teaching issues regarding the setup of the course and the practical and summarize student and teacher experiences. In the scientific part, the two student teams (Team I: Nikolai, Sebastian, Daniel; Team II: Sarah, Pierre) present their solutions for implementing efficient and numerically stable implementations of the TKF91 algorithm. The two teams worked independently on implementing the same algorithm. Hence, since the implementations yield identical results —with slight numerical deviations— we are confident that the implementations are correct. We describe the optimizations applied and make them available as open-source codes in the hope that our findings and software will be useful to the community and similar programming practicals at other universities.

1 Introduction

In [7], Thorne, Kishino, and Felsenstein presented a method for pair-wise alignment of DNA sequences using a maximum likelihood (ML) approach. To this end, they developed an explicit statistical model of evolution that uses statistical insertions, deletions, and substitutions of nucleotides as basic operations for comparing two DNA sequences. An evolutionary model that is given as input in principle determines at which rate these three evolutionary events occur. These are then used to design a dynamic programming algorithm that computes the ML pair-wise alignment between two DNA sequences.

As substitution model, we assume the standard F81 stochastic model of nucleotide substitution [4]. The algorithm computes the optimal maximum likelihood sequence alignment using three matrices, one for each evolutionary event

(i.e., substitution, deletion, and insertion). The algorithm is given by the following dynamic programming recurrence:

$$M^0(i,j) = \frac{\lambda}{\mu}\pi_{a_i}\overline{p_0}(t)max\{M^0(i-1,j), M^1(i-1,j), M^2(i-1,j)\}$$

$$M^1(i,j) = \frac{\lambda}{\mu}\pi_{a_i}max\{P_{a_i \to b_j}(t)p_1(t), \pi_{b_j}\overline{p_1}(t)\}$$

$$max\{M^0(i-1,j-1), M^1(i-1,j-1), M^2(i-1,j-1)\}$$

$$M^2(i,j) = \pi_{b_j}\lambda\beta(t)max\{M^1(i,j-1), M^2(i,j-1)\}$$

Where λ and μ denote the birth and death rate, $\pi_x, x \in \{A, C, G, T\}$ denotes the equilibrium frequency of the nucleotides, $P_{x \to y}$ the transition probability from x to y, and $\overline{p_n}(t)$ the probability that after time t a so-called mortal link has exactly n descendants. As already mentioned, the values for t, λ, μ, and π are given as input. The values $\overline{p_n}$ can be pre-computed in constant time. For more details, please refer to the original paper [4] and the on-line task specification at `http://sco.h-its.org/exelixis/web/teaching/practical15/description/tkf91.pdf`.

To the best of our knowledge, there is a limited amount of related work on optimizing the algorithm. In 2000 Hein *et al.*[5] presented several techniques for optimizing the algorithm. However, their focus was slightly different. One main goal of the work was to conduct a ML estimate of the model parameters. This was achieved by implementing a banded DP that only fills DP cells within a band along the diagonal of the DP matrix. Evidently, this can yield suboptimal results, but was deemed sufficient to estimate the ML parameters. Once this was done, the authors performed a full, unbanded DP computation with the previously computed ML parameter estimates. Note that, optimizing ML model parameters or using banded approaches was not part of the task specification for the students. Finally, the authors propose similar simplifications of the recursion as we do here, with the only difference that they use the term tabulation instead of lookup table or memoization we use here. Also, the authors do apparently not use a transformation into log space as we do here. Unfortunately, it was not possible to obtain a copy of the code to conduct comparisons to our implementations (pers. comm with J. Hein in August 2015).

In addition there also exists an R package (see `http://cran.us.r-project.org/web/packages/TKF/TKF.pdf`) that implements the TKF91 model, but only for amino acid data. We did therefore not compare it to our implementations.

TODO ALL (including Tomas!): is there anything else we should mention, please search a bit?

In the rest of the paper we refer to the dynamic programming paradigm as DP, and to dynamic programming matrices as DPM. The source codes developed by teams I and II are available for download at `https://github.com/nbaudis/bioinf2015` under the GNU GPL license.

The remainder of this paper is organized as follows: In Section 2, we describe the teaching setup and goals. The teams then present their implementations in Sections 3 and 4, respectively. The corresponding experimental results by both

teams are presented in Section 5. In the following Section 6, we summarize our teaching experiences. We conclude in Section 7.

2 Teaching Perspective, Goals and Course Outline

2.1 Teaching Setup & Goals

Courses at the Master level in our computer science department are organized in so-called modules over two semesters. In the first semester of the Bioinformatics module, we teach a lecture called "Introduction to Bioinformatics for Computer Scientists", since KIT does not offer a stand-alone Bioinformatics degree. This lecture covers basic topics such as an introduction to molecular biology, classic pair-wise sequence alignment, BLAST, de novo and by-reference sequence assembly, multiple sequence alignment, phylogenetic inference, MCMC methods, and population genetics.

In the second semester of the module, students can choose if they want to do a seminar presentation or the programming practical whose results we describe here. The goal of the practical is to carry out a self-contained project and write, as well as release software, that will be useful to the evolutionary biology community. Another key focus is on using tools (e.g., static analyzers, memory checkers) that increase software quality. Note that, at a CS department, designing "classic" bioinformatics analysis pipelines using scripting languages is typically not considered as "real programming" by the students. Hence, we needed to define a project that required coding in `C/C++` or `Java`. One should also strive to avoid having the students extend existing software, since this is generally frustrating and hinders creativity.

We thus decided to ask the students to implement efficient, sequential versions of the TKF91 algorithm that can also be used as library routines. Since the DP algorithm as such, is relatively straight-forward to implement for a computer science student, the main focus was on code optimization. The students were thus asked to implement a highly efficient version of the code in |C— or |C++— using all capabilities of a modern CPU (e.g., SSE3 and AVX intrinsics). The main motivation for this, was to give the students enough time to experiment with low-level optimization strategies on modern CPUs. Moreover, this project allowed to apply a broad range of skills acquired in the Bioinformatics and other master-level modules at our department. Furthermore, the TKF91 model required understanding and applying the discrete pair-wise sequence alignment methods and likelihood-based models for sequence evolution introduced in the lectures. To foster competition among the teams, an award (dinner payed by A.S.) was announced for the team that would implement the fastest code. To allow for a fair comparison of the codes and the CPU-specific optimizations we provided the students access to a reference machine with 4 physical cores (Intel i7-2600 running at 3.40GHz) and 16GB RAM.

In terms of project documentation, students are usually required to write a report. However, in the present case, we jointly took the decision to write a paper

about the practical and upload it to |biorxiv—. This has the positive effect that students also learn how to write scientific papers.

2.2   Code Quality Assessment

In order to continuously monitor and improve the quality of our source codes, we used several tools and methods throughout our project. We deployed both, static, as well as dynamic code analysis tools.

Static Analyses Static analyses help to identify programming errors such as incorrect programming language syntax or typing errors. We compiled our codes with the `gcc` compiler using *all* available and reasonable warning flags[1]. This allowed us to identify potential programming errors at an early stage. Due to its more pedantic nature (i.e., ability to detect more errors), we also used the `clang` compiler with respective flags[2] periodically alongside of `gcc` to further reduce the amount of potential programming errors. Note that, |clang— conducts a static code analysis.

Dynamic Analyses These analyses cover runtime issues, mainly memory leaks. We used `valgrind` and its sub-module `memcheck` for detecting memory-related errors in our programs.

3   Implementation of Team I

Before describing our vectorized implementation, we will cover the necessary prerequisites. In Section 3.1 we describe the DP algorithm in more detail as well as the corresponding wave-front parallelism we exploited in our implementation. In Section 3.2, we describe our memory layout concept, which is required to improve the efficiency of the vectorization. Thereafter, in Section 3.3, we describe how we cache intermediate computations. Then, we explain our vectorization approach in Section 3.4. Section 3.5 outlines our considerations regarding the memory alignment of the data that constitutes an important part of the vectorized implementation.

3.1   Dynamic Programming and Wave-front Parallelism

DP is a widely-used technique to efficiently solve a class of, at first sight, apparently hard computational problems when they exhibit `overlapping subproblems` and `optimal substructure`. Pair-wise sequence alignment falls into this class.

---

[1] -Wall -Wextra -Wredundant-decls -Wswitch-default -Wimport -Wno-int-to-pointer-cast -Wbad-function-cast -Wmissing-declarations -Wmissing-prototypes -Wnested-externs -Wstrict-prototypes -Wformat-nonliteral -Wundef

[2] -Weverything -pedantic

Another advantage of DP is that it is comparatively straight-forward to parallelize computations along the anti-diagonals of the DPM. This DPM parallelization approach is known as wave-front parallelism. The underlying idea is that matrix entries along the same DP anti-diagonal d can be computed independently from each other (and hence in parallel) if the preceding anti-diagonal $d-1$ has been computed. Therefore, we can deploy vector intrinsics to accelerate calculations along anti-diagonals.

3.2   Memory Layout

We first introduce our memory layout for the three DP matrices (M^0, M^1, M^2) since it is performance-critical. Our vectorization needs to attain high data locality to efficiently use the CPU cache. To this end, we permuted the indexing scheme of the DP matrix such that neighboring entries on an anti-diagonal are stored contiguously in memory. Figure 1 depicts the memory mapping of the matrix. It is layed out neither in a column-major nor row-major fashion, but stored linearly by anti-diagonals.
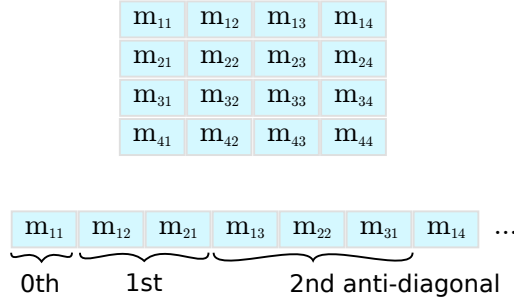

Fig. 1. DPM memory layout

We index the matrices using a modified version of the `Cantor pairing function`, where a tuple is assigned to an offset. The original Cantor pairing function π assigns an integer to a pair of integers (e.g., $\pi : (\mathbb{N}, \mathbb{N}) \to \mathbb{N}$). Since the size of the DPM is given by the length of the input sequences and because it is not infinite, we modified the pairing function to accept a tuple of two integers from the interval [0..length(Sequence 1|2)] as input. As a consequence, the DPM is divided into three parts: the *opening*, *intermediate*, and *closing part*. In the opening part, the modified pairing function is identical to the original function; in the intermediate and closing parts, the modified pairing function is calculated by subtracting an offset from the original pairing function. We calculate the offset via the original Cantor pairing function.
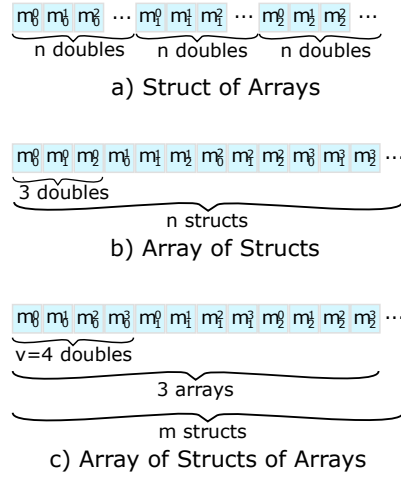
In Figure 2, we show the index calculations for the anti-diagonals of a 5×3-matrix. The blue frame denotes the actual boundaries of this 5×3 DP matrix, the blue cells represent the opening part, the green cells inside the frame the

0	1	3	6-0	10-1	0	1	3
2	4	7-0	11-1	16-3-1	2	4	
5	8-0	12-1	17-3-1	23-6-3	5		
0	1	3	6				
2	4	7					
5	8						

Fig. 2. Indexing scheme for anti-diagonals

intermediate part, and the red cells inside the frame the closing part. Green and red cells outside the matrix boundaries represent the offset calculation for the intermediate and closing part. For the DP calculations, only indices that correspond to DPM entries are required.

Given the anti-diagonal indexing scheme for a single matrix, we can now devise the memory layout for the three DP matrices (M_0, M_1, M_2). This is because calculating a DP value requires accessing values from all three matrices, which are located in different memory regions. To improve data locality while, at the same time, using efficient vector load and store operations, the data has to be arranged accordingly. Since we are using double-precision floating-point numbers for all calculations, an SSE3 vector (128 bit) can hold 2 values, while an AVX vector (256 bit) can hold 4 values. To this end, we evaluated the following three alternative matrix layouts (see Figure 3):

m_0^0 m_0^1 m_0^2 $\cdots$ m_1^0 m_1^1 m_1^2 $\cdots$ m_2^0 m_2^1 m_2^2 $\cdots$
$\underbrace{\hphantom{m_0^0 \ m_0^1 \ m_0^2}}_{n\ doubles}$ $\underbrace{\hphantom{m_1^0 \ m_1^1 \ m_1^2}}_{n\ doubles}$ $\underbrace{\hphantom{m_2^0 \ m_2^1 \ m_2^2}}_{n\ doubles}$

a) Struct of Arrays

m_0^0 m_0^1 m_0^2 m_1^0 m_1^1 m_1^2 m_2^0 m_2^1 m_2^2 m_3^0 m_3^1 m_3^2 $\cdots$
$\underbrace{\hphantom{m_0^0 \ m_0^1 \ m_0^2}}_{3\ doubles}$
$\underbrace{\hphantom{m_1^0 \ m_1^1 \ m_1^2 \ m_2^0 \ m_2^1 \ m_2^2 \ m_3^0}}_{n\ structs}$

b) Array of Structs

m_0^0 m_0^1 m_0^2 m_0^3 m_1^0 m_1^1 m_1^2 m_1^3 m_2^0 m_2^1 m_2^2 m_2^3 $\cdots$
$\underbrace{\hphantom{m_0^0 \ m_0^1 \ m_0^2 \ m_0^3}}_{v=4\ doubles}$
$\underbrace{\hphantom{m_0^0 \ m_0^1 \ m_0^2 \ m_0^3 \ m_1^0 \ m_1^1}}_{3\ arrays}$
$\underbrace{\hphantom{m_0^0 \ m_0^1 \ m_0^2 \ m_0^3 \ m_1^0 \ m_1^1 \ m_1^2 \ m_1^3}}_{m\ structs}$

c) Array of Structs of Arrays

Fig. 3. Alternative data layouts for storing anti-diagonals

Struct of Arrays (SoA) (`struct { double m0[n], m1[n], m2[n]; } data;`) This layout allows for simple vector load and store operations. However, the three DPMs are located in separate memory regions, which decreases data locality and hence cache efficiency.

Array of Structs (AoS) (`struct { double m0, m1, m2; } data[n];`) This layout exhibits improved data locality by grouping values closely together that are – in most cases – accessed simultaneously. On the other hand, loading and storing vectors requires disentangling and interleaving them, which requires several, potentially costly, vector shuffle operations.
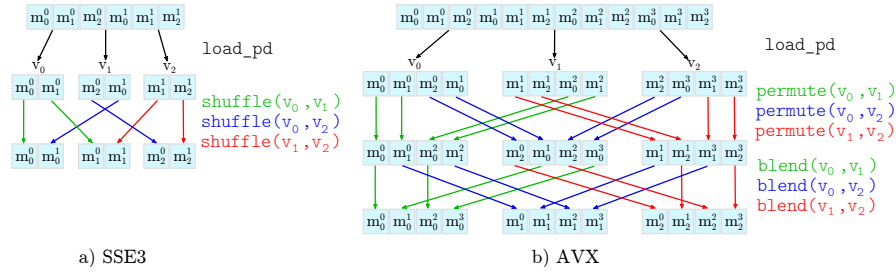
Fig. 4. Disentangling operations for a) SSE3 (128 bit) and b) AVX (256 bit) vectors.

Figure 4 illustrates these shuffle operations. For m_x^y, the subscript x indicates the index of the matrix and the superscript y the SSE3/AVX vector element index y. The first row of the Figure depicts the *AoS* data layout. Initially, we perform a vector `load` operation to load the data into temporary vector registers. Thereafter, we shuffle the temporary vectors such that (i) each vector only contains elements from one of the three DP matrices and (ii) the individual elements are in the correct order with respect to the anti-diagonal. For both, SSE3 as well as AVX instructions, we need to perform three `load` operations and three `shuffle`/`permute` operations. For AVX, three additional `blend` operations are required to correctly order the data elements.

Array of Structs of Arrays (AoSoA) (`struct { double m0[v], m1[v], m2[v]; } data[m];`) This layout stores vectors such that adjacent elements for each of the three matrices can be directly loaded into a SSE3/AVX vector. While solving both aforementioned problems (low data locality and costly vector operations), this complicates DP cell accesses above the current anti-diagonal. The required values above the current anti-diagonal are not stored contiguously and therefore we either need to use `shuffle`/`permute` operations or conduct partial loads (i.e., load single elements into vectors).

All three layouts exhibit different performance characteristics with respect to cache efficiency and complexity of the load, store, as well as shuffle operations.

We implemented prototypes for all three approaches to analyze their performance and found that the AoS approach performs best. Note that, the computational cost of the `shuffle` operations outweighed the improved cache efficiency in the AoS and AoSoA layouts (also see Table 5.2 in Section 5.2).

3.3   Memoization

The [7] model has several parameters. Thus, it initially seemed inevitable to carry out the non-trivial computations for filling DP cells from scratch for each individual cell. However, after analyzing which expressions are constant and can thus be pre-computed and reused (i.e., memoized), we reduced the operations required for calculating a DP cell value to just three: indirection, summation, maximum.

Obvious savings can be achieved for expressions that only depend on the time parameter t, the birth rate λ, and the death rate μ. These values are constant and given as input parameters. We also observed that the cell updates in the DP algorithm only require a constant number of common sub-calculations/components: When the nucleotide states at the current indices for the two sequences are available, the score (cell value) can be computed by only using the current nucleotide pair and the neighboring values in the three DP matrices.

The memoization scheme (lookup table) for all possible configurations is shown in Figure 5. The penalty functions C^i take one or two nucleotide states as parameters. Thus, these penalties can be stored in a memoization table of 4 and 16 entries, respectively. Using only 24 memoized values, we were able to substantially simplify and accelerate the cell updates. In addition, this simplification now allows to apply the logarithm for preventing numerical underflow. This was not possible before, since taking the logarithm of the original equation would have been too expensive computationally. In addition, vectorizing the cell updates is simpler, since the remaining calculations are less complex. Finally, this simplification reduces the number of possible memory layout and vectorization options.

At a later point of the project, it became evident that the indexed loads from the memoized penalty matrices C^0, C^1, C^2 as shown in Figure 5 caused a performance degradation. To address this problem, we tried to pre-compute a larger lookup table of vector-sized elements, that we indexed by a specific nucleotide permutation. Consider the following example for AVX vector intrinsics. We need to calculate a bijective mapping for a set of four nucleotides to an integer representing one of the $4^4 = 256$ possible permutations (e.g., a perfect hash) for C^0 and C^2, and one of the possible $4^{2 \cdot 4} = 65536$ permutations for C^1 respectively. The mapping is implemented as base conversion from the set of 4 nucleotides to an unsigned integer via appropriate bit operations.

While this approach has exponential space requirements as a function of the vector width, using a lookup table of $65536 \cdot 32$ bytes = 2 MB for C^1 for the AVX version of our code was still feasible. However, the performance evaluation revealed that too much time is spent to populate the table. Thus, we abandoned this approach.

$$C^0(a) = \frac{\lambda}{\mu}\pi_a\overline{p_0}(t)$$

$$C^1(a,b) = \frac{\lambda}{\mu}\pi_a max\{P_{a\to b}(t)p_1(t), \pi_b\overline{p_1}(t)\}$$

$$C^2(b) = \pi_b\lambda\beta(t)$$

$$M^0(i,j) = C^0(a_i)max\{M^0(i-1,j), M^1(i-1,j), M^2(i-1,j)\}$$

$$M^1(i,j) = C^1(a_i,b_j)max\{M^0(i-1,j-1), M^1(i-1,j-1), M^2(i-1,j-1)\}$$

$$M^2(i,j) = C^2(b_j)max\{M^1(i,j-1), M^2(i,j-1)\}$$

Fig. 5. A re-formulation of the DP step using the memoized sub-problems (lookup tables) C^i.

3.4   Vectorization

We implemented a vectorized version for computing M_0, M_1, and M_2 using `add`, `max`, `load`, and `store` operations for both SSE3 and AVX instructions. With these operations, we can process n matrix elements in parallel. Depending on the location of the vector on the anti-diagonal that is being processed, there will be $0 < n \le V$ valid elements to operate on per vector (with vector size $V := 4$ for AVX and $V := 2$ for SSE3).

Usually, data that has to be loaded into vectors need to be aligned, that is, the starting address of the vector data needs to be a multiple of 16 or 32 bytes. This memory alignment allows to use the aligned versions of the vector `load` and `store` operations, which are faster than the respective unaligned operations. Although we ensured memory-aligned accesses to the majority of the data (as will be explained in Section 3.5), we consistently used unaligned vector `load` and `store` operations (`loadu_pd` and `storeu_pd`) in the SSE3 and AVX versions of our code. This does not deteriorate performance, since we did not observe a significant performance degradation when applying unaligned load intrinsics on aligned data.

To prevent our `store` operations from overwriting data at the boundaries of the matrices, we ensured that for vectors with size $n < V$ (i.e., not completely filled/padded vectors) only n elements are written back into the matrices. For the SSE3 version, this can be achieved by only writing the lower part of the vector if its size is 1. For AVX, however, we covered all cases where $n < V$ holds using the `maskstore_pd` intrinsic and an appropriate mask for all possibles values of n (i.e., 3, 2, and 1).

3.5   Data Alignment

When vectorizing along the anti-diagonals, accessing the memory in the DPMs when they are stored in row-, or column-major order prevents deploying efficient vector operations for loading the values from the matrices into vector registers.

For such a matrix layout, the performance of the inner DP loop becomes heavily memory-bound. Therefore, we used the aforementioned layout by consecutive anti-diagonals together with the struct of arrays approach (see section 3.2), such that unaligned `load` and `store` operations for moving data directly to/from the anti-diagonal into vector registers can be utilized.
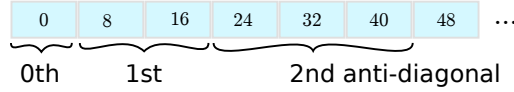

Fig. 6. Shift of the byte offset of the anti-diagonals

Figure 6 shows why a 16/32 byte alignment for the starting addresses of the anti-diagonals can not be achieved when just storing anti-diagonals linearly. To solve this problem, we used padding, that is, we allocate some dummy entries such that each anti-diagonal starts at a 16/32 byte-aligned address. As a consequence we also had to modify the indexing function from Section 3.2 accordingly. The main idea is to store each anti-diagonal starting at a 16/32 byte-aligned address.

We implemented a scheme where the row with index 1 is memory-aligned, because the row with index 0 is initialized *prior* to entering the DP loop. As a consequence, each anti-diagonal is aligned to an "odd" address ($a - 8$, where a is the desired byte alignment, for instance, $32 - 8 = 24$ bytes for AVX intrinsics).
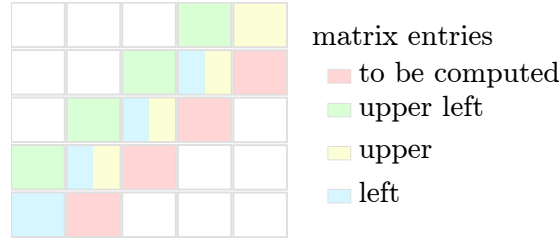

Fig. 7. Outline of the data accesses required for calculating an anti-diagonal

In Figure 7, we show a snapshot of the required values for computing the anti-diagonal elements with AVX intrinsics (vector length: 4 double precision values). In the current step we want to calculate the red elements. For calculating a single element, we require three elements from the two previous anti-diagonals. We need to access the element directly above (yellow) and left (blue) to the current element as well as the diagonal element above and to the left (green). The Figure illustrates that —omitting the asymptotically irrelevant boundary cases— one of the following four conditions holds:

1. red and blue start on an aligned address, yellow and green do not
2. red and yellow start on an aligned address, blue and green do not
3. yellow and green start on an aligned address, red and blue do not
4. red and green start on an aligned address, yellow and blue do not

For the computation, we need to perform three `load` operations in total for the blue, yellow, and green elements as well as one `store` operation for the red elements. Since, based on empirical observations, *unaligned* `store` operations require more time than unaligned load operations, we ensured that either condition 1 or 2 above is always met. Thus, we implemented our code such that, the first condition is met for the opening and intermediate part and the second condition holds true for the closing part of the DP matrices.

4   Implementation of Team II

In the following, we first describe how we transformed the algorithm into log-space to prevent numerical underflow. This also allowed us to simplify the formulas. Thereafter, we report how we improved data locality by storing the matrix entries in a dedicated data structure. While we experimented with different vectorization techniques, it turned out that the fastest code did not rely on vectorization. We thus conclude that, we managed to simplify the sequential code to such an extent, that, the potential advantages of using vector instructions are amortized by the additional overhead (see Section 4.3) we need for vectorization.

4.1   Mathematical Optimization - Basic Version

Numerical Underflow Prevention As the TKF91 algorithm performs successive multiplications of floating point numbers, preventing numerical underflow *is* a major issue. Underflow can occur, even for short input sequences with less than 100 nucleotides each. To address this issue we transformed all computations into log-space, to add logarithms of probabilities instead of multiplying probabilities. While it is still possible to experience numerical problems, even after this transformation, we expect that this is highly unlikely to occur for practical (empirical) input data. We did not observe any numerical issues for the range of input parameter values and sequence lengths (up to and including $10,000$ nucleotides) we tested.

Simplifying the Formulas We were able to omit redundant computations be re-using already computed values from previous matrix entries. For example, we observed that

$$M^0(i+1,0) = M^0(i,0) + \log(\gamma_{i+1}) + \log(\zeta_{i+1}) + \log(\beta(t)) + \log(\pi_{a_{i+1}}) + \log(\bar{p_0}(t)).$$

After replacing $\beta(t), \bar{p_0}(t), \gamma_i$, and ζ_i by their respective formulas, we noted that some terms appear multiple times. Operating in log-space allowed us to further

simplify the formulas. Especially the logarithmic rules $\log(a*b) = \log(a) + \log(b)$ and $\log(a/b) = \log(a) - \log(b)$ allowed us to replace multiplications and divisions with additions and subtractions.

In the following formulas, $1 \leq i \leq n$ and $1 \leq j \leq m$, if not noted otherwise.

Matrix Initialization

$$M^0(0,0) = -\infty$$

$$M^1(0,0) = \log(\gamma_0) + \log(\zeta_1) = \log(1 - \frac{\lambda}{\mu}) + \log(1 - \lambda * \beta)$$

$$M^2(0,0) = -\infty$$

$$M^0(1,0) = \log(\gamma_1) + \log(\zeta_1) + \log(\bar{p}_0) + \log(\pi_{a_1})$$

$$= \log(1 - \frac{\lambda}{\mu}) + \log(\lambda) + \log(1 - \lambda * \beta) + \log(\beta) + \log(\pi_{a_1})$$

$$M^0(i,0) = M^0(i-1,0) + 2*\log(\lambda) + 2*\log(\beta) + \log(\pi_{a_i}), i \geq 2$$

$$M^1(i,0) = -\infty$$

$$M^2(i,0) = -\infty$$

$$M^0(0,j) = -\infty$$

$$M^1(0,j) = -\infty$$

$$M^2(0,1) = \log(\gamma_0) + \log(\zeta_2) + \log(\pi_{b_1})$$

$$= \log(1 - \frac{\lambda}{\mu}) + \log(1 - \lambda * \beta) + \log(\lambda) + \log(\beta) + \log(\pi_{b_1})$$

$$M^2(0,j) = M^2(0,i-1) + \log(\lambda) + \log(\beta) + \log(\pi_{b_i}), j \geq 2$$

Further Initialization

$$M^0(i,j) = \log(\lambda) + \log(\beta) + \log(\pi_{a_i})$$

$$M^1(i,j) = \log(\lambda) - \log(\mu) + \log(\pi_{a_i}) + \log(\max\{P_{a_i \to b_j} * \bar{p}_1, \pi_{b_j} * \bar{p}_1\})$$

$$= \log(\lambda) - \log(\mu) + \log(\pi_{a_i}) + \log(1 - \lambda * \beta) + \max \begin{cases} \log(P_{a_i \to b_j}) - \mu * t, \\ \log(\pi_{b_j}) + \log(1 - e^{-\mu * t} - \mu * \beta) \end{cases}$$

$$M^2(i,j) = \log(\lambda) + \log(\beta) + \log(\pi_{b_j})$$

Dynamic Programming Step

$$M^0(i,j) = M^0(i,j) + \max\{M^0(i-1,j), M^1(i-1,j), M^2(i-1,j)\}$$

$$M^1(i,j) = M^1(i,j) + \max\{M^0(i-1,j-1), M^1(i-1,j-1), M^2(i-1,j-1)\}$$

$$M^2(i,j) = M^2(i,j) + \max\{M^1(i,j-1), M^2(i,j-1)\}$$

Pre-computing the Logarithms In general, replacing a multiplication by two logarithms and an addition is more costly. However, we managed to circumvent the additional computational cost of log space. This is because our formulas above contain only 25 different logarithm invocations that only depend on given, constant input parameters. Hence, all these logarithms can be pre-computed.

A major disadvantage of logarithmic transformations is that errors in the logarithm computation accumulate over a sequence of operations. To further investigate this, we used the high precision mathematical library of the `boost`-framework [1]. It offers datatypes that dynamically adapt their floating point precision to avoid numerical errors as well as under-/overflow. Our initial idea was to use this library only for pre-computing the logarithms, and thereby reduce the induced runtime overhead. However, converting these arbitrary precision datatypes back into standard double precision floating point values introduced additional loss of precision **TODO: with respect to the standard log function?**, when applying the respective conversion functions **TODO: please name the conversion functions you used!**. Therefore, we abandoned this path.

We denote the code obtained by applying the above techniques and transformations as the basic version.

4.2   Matrix Storage Schemes - Improved Version

Matrix as Array In the first, naïve implementation, we stored each matrix (M^0, M^1, M^2) in a separate array, using row-major order. The matrix entry at position (i, j) is stored at index position $i * (m + 1) + j$ in the array (see Figure 10).

To illustrate the shortcomings of this approach, consider the following example: assume that six matrix rows fit into one cache line. Then, performing one iteration of the inner DP loop requires loading three cache lines, one per DPM. This is depicted in part a) of Figure 8. If we further assume a cache capacity of only two cache lines, every iteration would then force one of the lines to be swapped out and decrease memory efficiency.

Array-of-Structs Data Structure The dynamic programming step (see Algorithm 1) accesses the matrices M^0, M^1 and M^2 at the same index position to determine the maximum entry. Thus, we improved data locality by storing the entries from the three matrices located at identical index positions contiguously in memory. For this, we implemented a data structure called `MatrixEntry` that consists of three double values: m_0, m_1 and m_2. Then, we used the Matrix as Array approach as before, but containing `MatrixEntry` structs as elements instead of simple `double` values (see Figure 9).

Let us now consider the example in Figure 8 again. With six matrix lines filling one cache line, a cache line now contains data from all three matrices. Consequently, all operations for a single DP cell update will access at most two cache lines. If we assume a cache capacity of two cache lines again, cache misses will now only occur when traversing row boundaries.
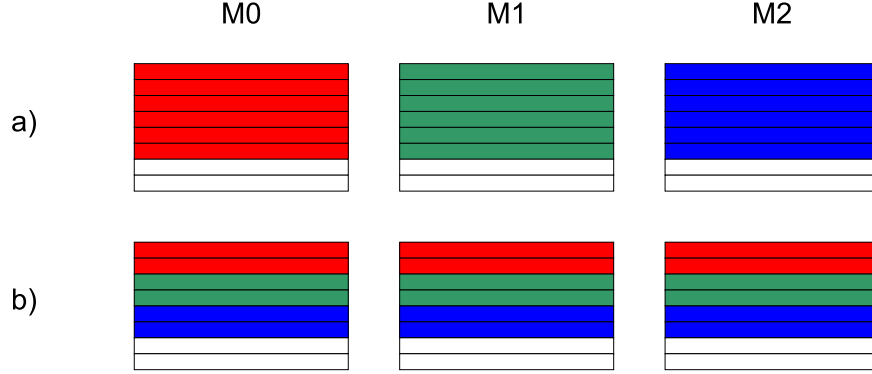
Fig. 8. Relation of cache lines and matrices, using the method of allocating each matrix separately(**a**)) and the Array-of-Structs method (**b**)). Each color represents a distinct cache line.

By using the `perf stat` tool, we found that using the `MatrixEntry` data structure indeed reduced the number of page faults which we use as a proxy for cache efficiency. **why didn't you use cachegrind then if perf-stat yielded variable results and only reprts page faults?**

Algorithm 1: The dynamic programming step, row-major version

---
1 ...// some initializations, see Appendix 4.1
2 **for** $i = 1, \ldots, n$ **do**
3   **for** $j = 1, \ldots, m$ **do**
4   ...// some initializations, see Appendix 4.1
5 $coord \leftarrow CO(i, j)$;
6 $up \leftarrow CO(i, j - 1)$;
7 $diag \leftarrow CO(i - 1, j - 1)$;
8 $left \leftarrow CO(i - 1, j)$;
9 $m[coord].m_0 \leftarrow m[coord].m_0 + \max\{m[left].m_0, m[left].m_1, m[left].m_2\}$;
10 $m[coord].m_1 \leftarrow$
             $m[coord].m_1 + \max\{m[diag].m_0, m[diag].m_1, m[diag].m_2\}$;
11 $m[coord].m_2 \leftarrow m[coord].m_2 + \max\{m[up].m_1, m[up].m_2\}$;
12   **end**
13 **end**

Alternative Storage Schemes Since each DP cell update needs to access the top, left, and upper-diagonal elements of the matrices, storing the matrix anti-diagonals linearly (see Figure 11 and considerations by Team I) will mini-
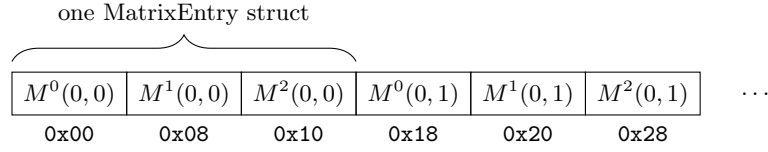
one MatrixEntry struct

$M^0(0,0)$	$M^1(0,0)$	$M^2(0,0)$	$M^0(0,1)$	$M^1(0,1)$	$M^2(0,1)$	$\cdots$
0x00	0x08	0x10	0x18	0x20	0x28	

Fig. 9. Array-of-Structs data structure for storing the three matrices in memory. Each `MatrixEntry` element stores the entries of a single index for all three matrices. The structs are stored contiguously in row-major order.

mize cache misses. Finding an inexpensive-to-compute closed formula that maps the index position (i, j) representing the row and column of a matrix to wave-front-coordinates turned out to be challenging (see also discussion by Team I in Section 3.2). The index of the diagonal is $i + j$, the sum of the row index and the column index. Determining the number of elements on the anti-diagonal and especially the correct position along the anti-diagonal proved more difficult though. The indexing formulas we tested were too computationally expensive and required more computations than the actual cell updates. As an alternative, we tried storing pre-computed index mappings in an additional matrix. However, this slowed down the program as computing the mapped indices required more arithmetic operations than the actual TKF91 algorithm. Figure 12 illustrates the main problem which has to be solved for efficiently indexing-diagonals: Given an wave-front index k, how do we obtain the indices for the upper, upper-left-diagonal, and left element of the matrix? While the required offsets are constant for a single anti-diagonal, they change for successive anti-diagonals.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Fig. 10. Row-major indexing

0	1	3	6	10
2	4	7	11	14
5	8	12	15	17
9	13	16	18	19

Fig. 11. Wave-front indexing

We denote the code obtained by applying the above storage scheme, the improved version.

4.3   Vectorization Attempts

During code development we conducted a partial vectorization for two of our implementations: the basic log-space implementation (Section 4.1) and the improved log-space implementation (section 4.2), using a more cache-efficient data structure to store the matrices (Section 4.2).

In both cases we mainly focused on vectorizing the code that initializes the three matrices. This is because, our analyses with `Valgrind callgrind` revealed
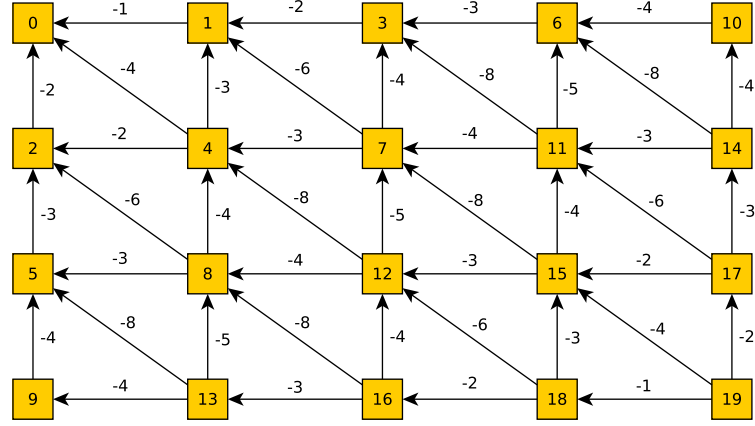
Fig. 12. Offsets in wave-front/anti-diagonal indexing

that this part of the code required $75 - 80\%$ of overall runtime. Additionally, as the initializations are completely independent of each other, this part should be straight-forward to vectorize.

Several code transformations were necessary to vectorize the code. First, we had to ensure that vector `load` and `store` operations are performed on correctly aligned memory addresses (see discussion by Team I). Secondly, we needed to devise a strategy for iterating over matrices and handling matrix sizes that are not multiples of the vector width. Finally, we needed to store calculation terms (e.g., $\log(\pi)$'s or $P_{a_i \to b_j}$'s), in appropriate vector types and had to replace all basic arithmetic operations by their vectorized counterparts.

Log-space implementation For the vectorization of our basic version (see Section 4.1), we implemented an appropriate memory alignment by using the `__attribute__(aligned(size))` attribute for stack allocations, and the `posix_memalign` function for heap allocations.

We also changed the iterations through the matrices to start at entry 0 of each row, as starting iterations at position 1 would result in memory accesses at incorrectly aligned addresses. Furthermore, as we store the matrix rows contiguously in memory, we had to pad the rows to ensure a correct alignment of the first entry in each row.

The strategy we chose to deal with row dimensions that are not multiples of the vector width is straight-forward. For each row, we compute as much as possible via vector intrinsics and the remainder sequentially.

Version using Array of Structs data structure While the strategy for dealing with row dimensions that do not fit vector widths remained the same

in the improved version (introduced in Section 4.2) of our program, we had to apply several modifications to be able to use vector intrinsics in conjunction with this more cache-efficient data structure.

As Figure 9 shows, using the Array-of-Structs scheme results in a non-contiguous storage of row entries. As a consequence, loading and storing vector registers is not straight-forward. To this end, we implemented load and store operations that fit our data structure. These operations rely on vector functions that load and store single double precision floating point values. An advantage of this is that we do not need to enforce correct data alignment any more. However, a single `_mm_load_pd` on contiguous memory is substantially faster than a series of `_mm_loadl_pd` and `_mm_loadh_pd` operations on non-contiguous memory locations.

Thoughts on wave-front vectorization Our two log-space versions of TKF91 reduce the DP cell update calculations to determining the largest of two or three values. In turn, this means that the relative amount of time spent for DP cell updates is almost negligible. Nevertheless, we invested time to explore potential vectorized wave-front versions of the program.

Initially, as Team I, we investigated appropriate DP cell storage and access schemes, to store the data contiguously with respect to the wave-front parallelization data access pattern. We found that, indexing a cell required 20-30 arithmetic operations as opposed to only 2-3 for the actual cell update. Thus, merely indexing a cell, using the standard row and column coordinates, requires an excessive amount of computations. Our preliminary benchmarks yielded absolutely no performance improvement for the vectorized versions of the basic as well as the improved versions of our code. Therefore, we abandoned the vectorized wave-front approach.

5   Evaluation and Testing

We initially describe the benchmark data we used (Section 5.1) as well as the experimental setup. Then we present the runtime results for teams I and II. We conclude with a thorough analysis of the impact of the logarithm implementation used in Section 5.4.

the perf results of team 1 and team 2, need to be completely re-written! AS will do the re-writing once he has the data. the logarithm section is fine.

5.1   Benchmarks Data

For testing and performance assessment, we used empirical benchmark datasets. We download multiple sequence alignments (MSAs) from `http://goo.gl/nlD4nb` that were used in [2].

As test platform we used the aforementioned reference hardware (see Section 2.1).

5.2   Results of Team I

From each of the six MSAs, we picked ten sequences **at random or how did you pick them???** and disaligned them. Then, we computed all possible 45 pairwise alignments between each set of ten sequences. We found that the median of five samples, where each sample consisted of the average runtime for ten `iterations` (i.e., $5 * 10 * 45 = 2250$ pair-wise alignments per dataset) yielded stable runtime estimates. As mentioned in Section 3.5, we performed various performance comparisons using this benchmark suite, such as the viability of different vector load/store schemes or the comparison between the SoA, AoS, and AoSoA storage schemes. We depict the results of the SoA versus AoS versus AoSoA runtime experiments in Table 5.2. As already mentioned in Section 3.5, the SoA approach performs best.

which parameters did you use as input for the stat. alignment model? would be useful to show an example of a command line that you used to be able to reproduce your results. The results section should always contain or link to all data necessary for reproducing the results! You should also state what exactly you measured, i.e., overall run-ime kernel time etc. We will need to re-write this anyway

ms per iteration (compared to best)	SoA	AoS	AoSoA
BDNF	8.30 (1.01)	**8.22 (1.00)**	8.51 (1.04)
cytb	**26.19 (1.00)**	29.07 (1.11)	29.65 (1.13)
RAG1	20.88 (1.02)	**20.56 (1.00)**	20.94 (1.02)
RAG2	**24.64 (1.00)**	25.09 (1.02)	25.83 (1.05)
RBP3	**35.26 (1.00)**	36.35 (1.03)	37.62 (1.07)
vWF	44.54 (1.01)	**44.10 (1.00)**	46.27 (1.05)

Table 1. Benchmark results for SoA, AoS and AoSoA (from left to right)

5.3   Results of Team II

We first assess the performance and impact on the output of distinct logarithm implementations (Section 5.4). Thereafter, we measure the runtime of our code in Section 5.5.

5.4   Different Logarithm Libraries

In addition to the standard `log` function (from C++ `<math.h>`), we also used the `crlibm` library by Daramy *et al.* [3]. It includes versions of the logarithm function, which allow to explicitly specify the type of rounding strategy, namely rounding up, down, towards zero or towards the nearest integer, (**for instance?**) to be used.

To assess the correctness of our alignments, we computed the edit distance between the alignments **how is the edit distance between alignments defined? This needs to be explained!** obtained by using different logarithm implementations. The edit distance is a metric to quantify the difference between two strings. It counts the edit operations, insertions, deletions and substitutions, required to transform one string into the other. Each edit operation is assigned a cost value, resulting in a weighted sum of operations that form the final value. We assigned a cost of 1 to insertions, deletions, and substitutions of single letters. For our tests, we aligned all possible pairs of sequences from the BDNF MSA of the aforementioned benchmark data (see Section 5.1).

For the remaining parameters, we used $\lambda := 1, \mu := 2, \pi := (0.27, 0.24, 0.26, 0.23)$ and $\tau := 0.1$. We computed edit distances between the calculated alignments and the reference alignment that uses the `Boost.Multiprecision` logarithm implementation.

In particular the `log_ru` function from the `crlibm` library, which explicitly rounds up, produces a notably different alignment compared to `Boost.Multiprecision` reference alignment (see Table 2). **the following sentence is inconsistent, first you say notably different and then highly similar?** The computed likelihood scores were highly similar and the differences in the alignments were minimal **between what and what?** as can be seen in Figure 13. We finally decided to use the `log_ru` function from the `crlibm` library because it yielded the same results as the reference implementation (`http://sco.h-its.org/exelixis/web/teaching/practical15/scaledCode/tkf91_scaling.tar.gz`) in most cases.

Logarithm Function	Average Edit Distance to Reference Alignment
`log` from `<math.h>`	2.15
`log_ru` from `crlibm`	33.15

Table 2. Average edit distances from the `Boost.Multiprecision` reference alignment for different logarithm implementations.

5.5 Runtime Measurements

As input sequences for measuring runtimes, we used four pairs of randomly generated sequences with 10, 100, 1000, and 10000 nucleotides.

We measured the execution times around the kernel of the program, that is, excluding any I/O required to load parameters and input sequences, or the output of the program. The allocation and initialization of the three matrices are part of the kernel and as such were included in out measurements. **mention if this included or excluded DP matrix init times** We executed the kernel multiple times for each dataset and subsequently averaged the runtimes. To balance accuracy and overall benchmark time (see Table 3) we used different number of kernel runs, depending on the dataset size.

Alignment using standard C++ header `<math.h>`
```
ACGACTAGTCA-GC-TACG-AT-CGA-CT-C-ATTCAACTGACTGACA-TCGACTTA
A-GAG-AGTAATGCATACGCATGC-ATCTGCTATT---CTG-CTG-CAGTGG--T-A
```

Alignment using `Boost.Multiprecision`
```
ACGACTAGTCA-GC-TACG-AT-CGA-CT-C-ATTCAACTGACTGACA-TCGACTTA
A-GAG-AGTAATGCATACGCATGC-ATCTGCTATT---CTG-CTG-CAGTGG--T-A
```

Alignment using `log_ru`
```
ACGACTAGTCA-GC-TACG-AT-CGA-CT-C-ATTCAACTGACTGACA-TCGACTTA
A-GAG-AGTAATGCATACGCATGC-ATCTGCTATTC---TG-CTG-CAGTGG--T-A
```

Alignment from reference implementation
```
ACGACTAGTCA-GC-TACG-AT-CGA-CT-C-ATTCAACTGACTGACA-TCGACTTA
A-GAG-AGTAATGCATACGCATGC-ATCTGCTATTC---TG-CTG-CAGTGG--T-A
```

Fig. 13. Numerical alignment differences for input parameters $\pi := (0.25, 0.25, 0.25, 0.25), \lambda := 1, \mu := 2, \tau := 0.1$

Length of Sequences	Number of Runs
10	10000
100	1000
1000	100
10000	10

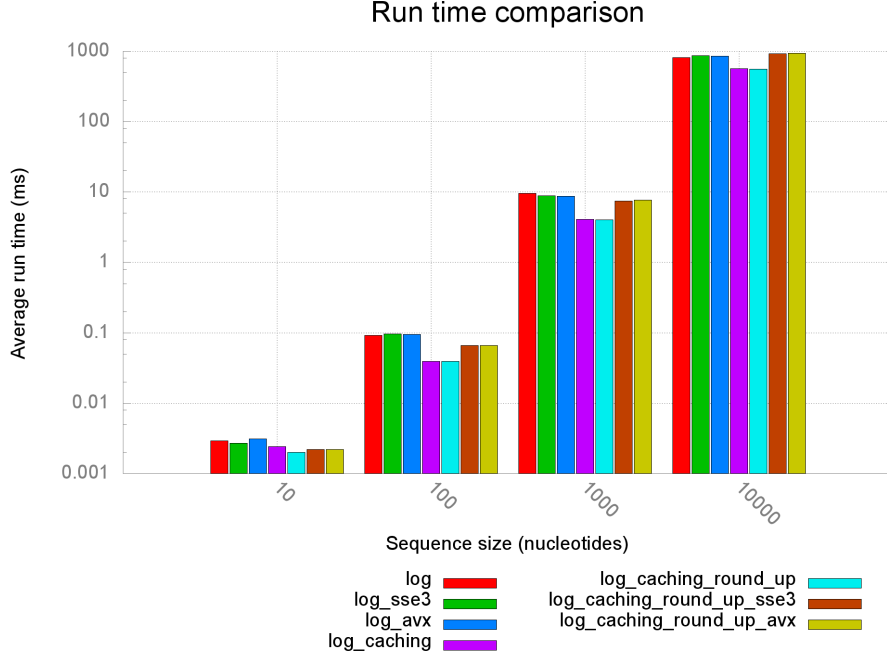**Table 3.** Number of runs per sequence length

Fig. 14. Comparison of average runtimes of different implementations. `sse3` and `avx` post-fixes denote the vectorized versions of the programs, using SSE3 and AVX vector intrinsics (Section 4.3). With `log` we denote the log-space transformed versions of TKF91 (Section 4.1). Versions using Arrays-of-Structs to store matrices (Section 4.2) are denoted by `caching`; `round_up` denotes versions relying on the `crlibm` library logarithm function `log_ru` (Section 5.4).

6 Teaching Results

6.1 What did Team I learn?

While the assigned task was straightforward and very manageable in principle, the implementation details were tricky. Preventing floating point underflow while optimizing performance at the same time represented a challenge. This is because, such technical problems are frequently ignored in other programming practicals at KIT that emphasize functionality. Furthermore, vectorizing with SIMD intrinsics required some rethinking and re-engineering because we needed to identify a suitable memory layout, optimize data alignment, and deal with boundary conditions (padding). While it was satisfying to address and solve problems progressively, there were always additional ideas to further improve the code (w.r.t. performance and design), which made prioritizing tasks important. We enjoyed the satisfaction of gaining yet another percent of execution time in combination with working with SIMD instructions that were not familiar to us.

The basic problem was clearly and outlined. There was enough freedom and time left to work on improving our solution. The scheduled project milestones turned procrastination and last minute work into a non-issue. Most surprisingly, we were always on schedule.

6.2   What did Team II learn?

For this assignment we had to tackle two major problems: preventing the loss of precision generated by numerical underflow and pursuing the elusive "most efficient" implementation. In the early stages, we came up with the idea of transforming the algorithm into log-space to solve the numerical issues. This did not only prove to represent an efficient solution, but also allowed us to further simplify the formulas and to pre-compute the vast majority of terms. In this context we also discovered deviations in the output alignments that depend on the specific implementation of the logarithm function being used. We therefore used the `crlibm` library of mathematical functions, that allowed us to assess the impact of logarithm functions with distinct rounding strategies on the final result.

To optimize the code we also used SSE3 and AVX intrinsics to vectorize the most work-intensive portions of the code. While we were unable to produce a faster vectorized code, we did learn how to use vector intrinsics and how to deal with the associated pitfalls. The largest performance gain was attained via a more cache-efficient data structure. For this, we used profiling tools such as `perf` and `clang` for the first time. Finally, we concluded that |C++— is appropriate for HPC projects, provided that, the problem and data-structures at hand are well understood.

7   Conclusion

We have implemented and made available two independent, highly optimized open-source implementations of the TKF91 algorithm for statistical pair-wise sequence alignment. We thoroughly assessed their performance and investigated aspects such as differing alignment results because of slight numerical deviations in logarithm implementations. The implementations by teams I and II are substantially different in the approach they take for optimizing program performance on standard x86 architectures. We proposed several methods for storing and addressing the three dynamic programming matrices. In addition, we show how the original equations can be simplified to (i) avoid numerical underflow issues and (ii) save a substantial amount of computations.

We make the source code available in the hope that, it will be useful to the community. In particular, some of the wave-front vectorization approaches might be applied to the more complex statistical alignment kernels used in programs such as BaliPhy [6].

Finally, the task and code at hand can be used to design similar bioinformatics programming practicals. For instance, one might ask students to try and come up with a faster implementation or use the existing implementations as library functions in some larger and more complex programming project.

References

1. Boost Multiprecision. `http://www.boost.org/doc/libs/1_59_0/libs/multiprecision/`. Accessed 08.09.2015.
2. Olaf RP Bininda-Emonds. transAlign: using amino acids to facilitate the multiple alignment of protein-coding DNA sequences. *BMC Bioinformatics*, 6(1):156, 2005.
3. Catherine Daramy, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, and Jean-Michel Muller. Cr-libm – a library of correctly rounded elementary functions in double-precision, 2004.
4. Joseph Felsenstein. Evolutionary trees from dna sequences: a maximum likelihood approach. *Journal of molecular evolution*, 17(6):368–376, 1981.
5. J Hein, Carsten Wiuf, B Knudsen, MB Møller, and G Wibling. Statistical alignment: computational properties, homology testing and goodness-of-fit. *Journal of Molecular Biology*, 302(1):265–279, 2000.
6. Marc A Suchard and Benjamin D Redelings. Bali-phy: simultaneous bayesian inference of alignment and phylogeny. *Bioinformatics*, 22(16):2047–2048, 2006.
7. Jeffrey Thorne, Hirohisa Kishino, and Joseph Felsenstein. An evolutionary model for maximum likelihood alignment of DNA sequences. *Journal of Molecular Evolution*, 33(2):114–124, 1991.