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.

1 Introduction

In [1], Thorne, Kishino, and Felsenstein presented a method for aligning DNA sequences using a maximum likelihood approach. To this end, they developed an explicit statistical model of evolution that includes insertions, deletions, and substitutions of nucleotides as basic operations for comparing two DNA sequences. A given evolutionary model assigns a weight to each of these three basic operations. These are then used to design a dynamic programming algorithm that computes the maximum likelihood pair-wise alignment between two DNA sequences.

As substitution model, we assume the standard F81 stochastic model of nucleotide substitution [**?**]. 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 follow-

ing 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 mortal link has exactly n descendants. The values for t, λ, μ, and π are given as input. The values for $\overline{p_n}$ can be pre-computed in constant time.

TODO related work

In the following we refer to the dynamic programming paradigm as DP, and to dynamic programming matrices as DPM.

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 **??**, 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 Alexis) was announced for the team that would implement the fastest code. To allow for a fair comparison of the codes and CPU-specific optimization 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 code, we used several tools and methods throughout our project. We deployed both, static, as well as dynamic code analysis tools.

Static Analysis Static analyses help to identify static 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 Analysis These analyses cover run-time issues, mainly memory leaks. We used `valgrind` and its sub-module `memcheck` for detecting memory-related errors in our programs.

[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

3 Implementation of Team I

Before describing our vectorized implementation, we 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 data alignment 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. 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.

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., $\pi : (\mathbb{N}, \mathbb{N}) \rightarrow \mathbb{N}$). 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,

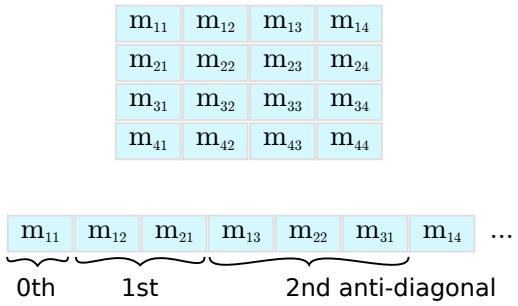

Fig. 1. DPM memory layout

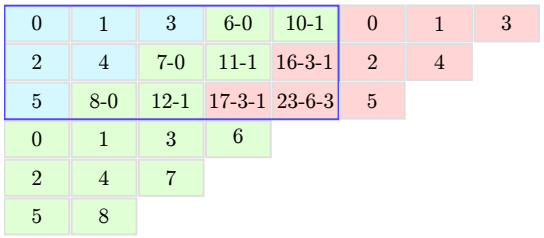

Fig. 2. Indexing scheme for the anti-diagonals

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

Given the anti-diagonal indexing scheme for a single matrix, we now need to 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 at different positions in memory. 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):

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 cache efficiency and data locality.

Array of Structs (AoS) (`struct { double m0, m1, m2; } data[n];`) This layout exhibits improved data locality by grouping values closely together that are –

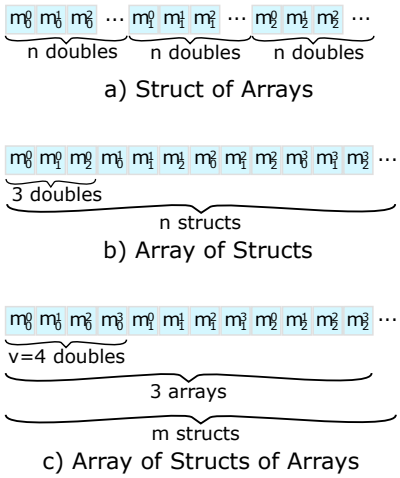

Fig. 3. Alternative data layouts for storing anti-diagonals

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.

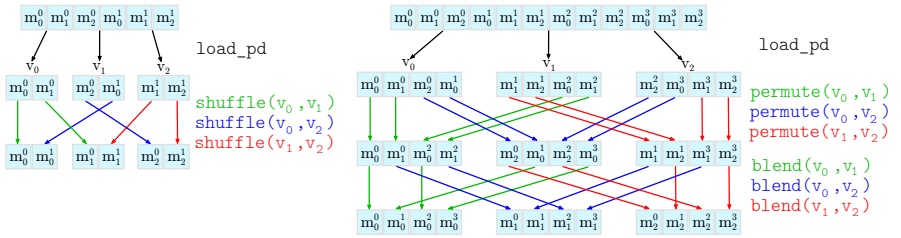

Fig. 4. Disentangling operations for a) SSE3 (128 bit) and b) AVX (256 bit) vectors.

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 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 **??**).

3.3 Memoization

The [1] 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 using only 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 penalties functions C^i that can be stored in lookup tables take one or two nucleotide states as parameters. Thus, the 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 defined 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

$$
\begin{aligned}
C^0(a) &= \frac{\lambda}{\mu}\pi_a\overline{p_0}(t) \\
C^1(a,b) &= \frac{\lambda}{\mu}\pi_a max\{P_{a\to b}(t)p_1(t), \pi_b\overline{p_1}(t)\} \\
C^2(b) &= \pi_b\lambda\beta(t) \\
M^0(i,j) &= C^0(a_i)max\{M^0(i-1,j), M^1(i-1,j), M^2(i-1,j)\} \\
M^1(i,j) &= C^1(a_i,b_j)max\{M^0(i-1,j-1), M^1(i-1,j-1), M^2(i-1,j-1)\} \\
M^2(i,j) &= C^2(b_j)max\{M^1(i,j-1), M^2(i,j-1)\}
\end{aligned}
$$

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

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 $4^{2\cdot4} = 65536$ 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 evaluation of the resulting performance revealed that too much time is spent to populate the table. Thus, we abandoned this approach.

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 $0 < n \leq V$ 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 needs 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 degrade 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 a row-, or column-major scheme 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 anti-diagonalized matrix layout together with a struct of arrays approach (see section 3.2), such that unaligned `load` and `store` operations for copying data directly from the anti-diagonal into vector registers can be utilized.

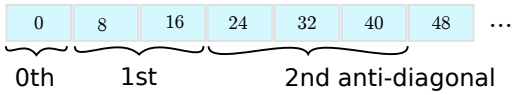

Fig. 6. Shift of the byte offset of the anti-diagonals

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).

In Figure 7, we show the required values for computing the inner DP loop for AVX (vector length: 4 double precision values). The elements marked in red are the ones we want to compute in the current step. For calculating a single element, we require three elements from the two previous anti-diagonals. The element directly above (yellow) and left (blue) to the current element and the diagonal element above and to the left (green) need to be loaded from the DP matrices. The Figure illustrates that (neglecting the asymptotically irrelevant boundary cases) one of the following four conditions holds:

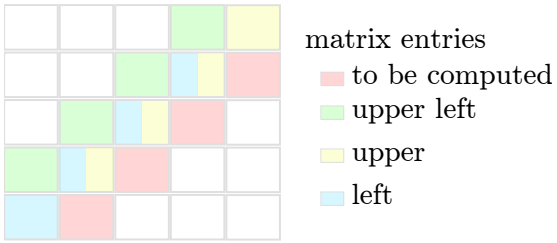

Fig. 7. Outline of the data required for an operation in the DP loop

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 and one `store` operation for the red elements. Since the *unaligned* `store` operation has greater relative performance impact than the unaligned load operations **why?Good question :) purely speculative, I guess. Although stores seem to take longer than loads.** , we ensured that either the first or the second of the above conditions is met. The first condition is implemented for the opening and intermediate part, the second condition holds true for the closing part of the DP matrices.

We achieved an appropriate alignment for each anti-diagonal by padding anti-diagonals as needed.

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 SIMD instructions are amortized by the additional overhead (**TODO add cross-ref to the section where you discuss this**) we need for vectorization.

4.1 Mathematical Optimization

Numerical Underflow Prevention As the TKF91 algorithm performs successive multiplications on 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 parameters 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 observed 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

$$\begin{aligned}
M^0(0,0) &= -\infty \\
M^1(0,0) &= \log(\gamma_0) + \log(\zeta_1) = \log(1-\frac{\lambda}{\mu}) + \log(1-\lambda*\beta) \\
M^2(0,0) &= -\infty \\
M^0(1,0) &= \log(\gamma_1) + \log(\zeta_1) + \log(\bar{p_0}) + \log(\pi_{a_1}) \\
&= \log(1-\frac{\lambda}{\mu}) + \log(\lambda) + \log(1-\lambda*\beta) + \log(\beta) + \log(\pi_{a_1}) \\
M^0(i,0) &= M^0(i-1,0) + 2*\log(\lambda) + 2*\log(\beta) + \log(\pi_{a_i}), i \geq 2 \\
M^1(i,0) &= -\infty \\
M^2(i,0) &= -\infty \\
M^0(0,j) &= -\infty \\
M^1(0,j) &= -\infty \\
M^2(0,1) &= \log(\gamma_0) + \log(\zeta_2) + \log(\pi_{b_1}) \\
&= \log(1-\frac{\lambda}{\mu}) + \log(1-\lambda*\beta) + \log(\lambda) + \log(\beta) + \log(\pi_{b_1}) \\
M^2(0,j) &= M^2(0,i-1) + \log(\lambda) + \log(\beta) + \log(\pi_{b_i}), j \geq 2
\end{aligned}$$

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 series of operations. To further investigate this, we used the high precision mathematical library of the `boost`-framework [**?**]. 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.

4.2 Matrix Storage Schemes

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, causing memory overhead.

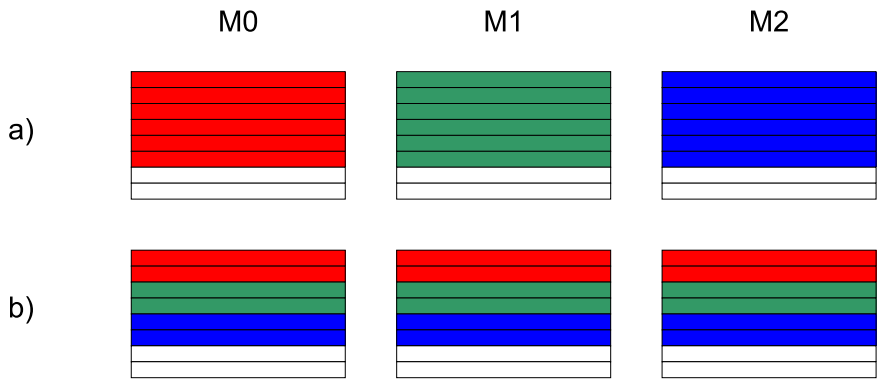

Fig. 8. Relation of cache lines and matrices, using the method of allocating each matrix separately(**a)**) and the Array-of-Structs method (**b)**). Each color represents a distinct cache line.

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 at the same index position 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 need to 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.

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?**

Alternative Storage Schemes Since each iteration in the dynamic programming step 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 minimize 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-

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

```
1  ...// some initializations, see Appendix ??
2  for i = 1, ..., n do
3  |  for j = 1, ..., m do
4  |  |  ...// some initializations, see Appendix ??
5  |  |  coord ← CO(i, j);
6  |  |  up ← CO(i, j − 1);
7  |  |  diag ← CO(i − 1, j − 1);
8  |  |  left ← CO(i − 1, j);
9  |  |  m[coord].m_0 ← m[coord].m_0 + max{m[left].m_0, m[left].m_1, m[left].m_2};
10 |  |  m[coord].m_1 ←
   |  |  m[coord].m_1 + max{m[diag].m_0, m[diag].m_1, m[diag].m_2};
11 |  |  m[coord].m_2 ← m[coord].m_2 + max{m[up].m_1, m[up].m_2};
12 |  end
13 end
```

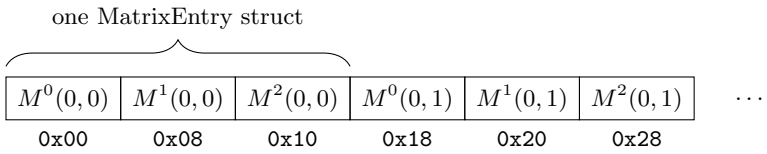

Fig. 9. Array-of-Structs data structure used to store the three matrices in memory. Each `MatrixEntry` element stores the entries of a single coordinate for all three matrices. The structs are stored contiguously in row-major order.

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 needed for 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

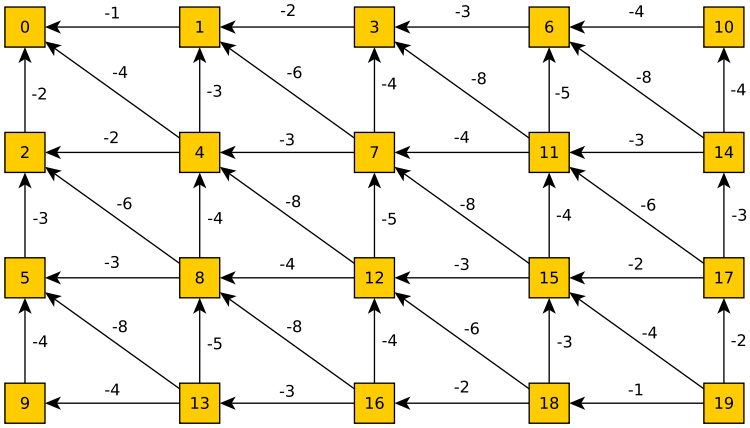

Fig. 12. Offsets in wave-front/anti-diagonal indexing

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, 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 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 needed to guarantee that vector `load` and `store` operations are invoked 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 the basic **did you properly introduce what the basic implementation is? Pierre: First paragraph in "Vectorization attempts" we name it as such, provide a back reference to 4.1. Maybe we could introduce it directly under the 4.1 section header? then back ref here again** Log-space implementation we implemented appropriate memory alignment by using the `__attribute__(aligned(`*`size`*`))`

attribute for stack allocations, and the `posix_memalign` function for heap allocations.

As a consequence, we also changed the iterations through the matrices to start at entry 0 of each row, as starting iteration at position 1 would result in incorrectly aligned memory accesses. Furthermore, as we stored the rows of the matrices contiguously in memory, we had to pad the row sizes to ensure the 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. Per 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 of our program, we had to apply several modifications to use vector intrinsics in conjunction with the 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, simply loading and storing vector registers cannot be used here. 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 correct data alignment is not required any more. However, we believe **why do you believe this, don't you have experimental evidence? Pierre: we didn't test them in isolation, but we assume that the load/store intrinsics that require contig. memory (_mm_load_pd) work faster than my function using _mm_loadl_pd and _mm_loadh_pd** that this approach also induces a significant performance overhead compared to load/store intrinsics that work on contiguous memory. **what are classically used operations? Pierre: changed it**.

Thoughts on wave-front vectorization Maybe somehow make the wave-front parts more consistent to avoid redundnacies? As the dynamic programming step of the TKF91 algorithm has data dependencies to the cells left of, above and diagonally left above of the cell to be computed, parallelization/vectorization requires a *wave-front* approach.

In a wave-front parallelization scheme, matrix entries are not computed by row or by column, but rather by computing the entries of each *anti-diagonal* in parallel. Crucially, the individual anti-diagonals have to be computed sequentially and in-order when dependencies exist between subsequent anti-diagonals. In this case, the current anti-diagonal is the front of the wave traversing the DP matrix.

The following sentence is too long and confusing, please re-formulate! Pierre: done The log-space **you use log-space inconssitently sometimes Log-spaca and sometimes as log-space** version of TKF91 reduces the dynamic programming step to finding the greatest of two to three values, depending on the matrix. This in turn meant, that the relative amount of time spent doing

so represents a negligible part of the overall algorithm. Nevertheless, we invested some time to devise a vectorized wave-front version of the program.

Initially, we attempted to find a suitable way for storing and accessing the matrix cells, to store data contiguously with respect to the wave-front data access pattern. We concluded that merely indexing **accessing or indexing? Pierre: I guess indexing is more specific, to me accessing encompassed indexing. changed it** a cell, using the usual row-, and column coordinates, requires a significant amount of computation by itself. Most importantly, indexing a cell **what? Pierre: changed it** requires significantly more arithmetic operations **please quantifdy! Pierre: we never found out, we abandoned trying to find a indexing formula when we realized that just accessing would require on the order of 20-30 operations,as opposed to the normal 2-3** than the actual computations conducted for each matrix entry.

Furthermore, our preliminary benchmarks had yielded absolutely no performance increase for the vectorized versions of either the basic log-space, or the enhanced, Array-of-Structs implementation**what are thos major im,plementations, maybe you should find better names for them Pierre: tried to name them**. As a consequence, we abandoned the wave-front vectorization approach.

5 Evaluation and Testing

5.1 Benchmarks and Test Suites

An automated test suite is extremely valuable for re-factoring and optimizing the code, in that it provides the confidence of being able to detect regressions early and get helpful hints on how to fix them. We used the *ctest* module of *CMake* to invoke a Python script driving our console wrapper through a number of inputs. As we also used a continuous integration service running the tests on new commits, we could easily spot when a change accidentally broke the build. This development process lead to a high level of satisfaction and faith to try out different ideas in separate branches, while immediately getting feedback.

TODO consistency

When the most obvious optimizations were done and we began to make use of vectorization instructions, we realized a benchmark suite based on the alignment data sets from the University of Oldenburg[3] using the Celero C++ benchmarking library[4] to have a reliable way of validating performance of our different approaches.

5.2 Results of Team I

From each of the six data sets, we picked ten nucleotide strings and computed all 45 pairwise alignments. We found that the median of five *samples*, each of which consisting of the average of ten such *iterations* (i.e. $5 * 10 * 45 = 2250$ alignments

[3] http://goo.gl/nlD4nb
[4] https://github.com/DigitalInBlue/Celero

per data set) gave reliable performance estimates. As stated in Section 3.5, we performed various performance comparisons based on this benchmark suite, such as the viability of different vector load and store schemes or the SoA, AoS, and AoSoA issue, of which the benchmark results on the reference hardware are depicted in Table 5.2. As already noted in Section 3.5, we deemed the SoA approach the most performant.

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

TODO

We conducted all measurements on a desktop machine with 16 GB RAM and a Intel^{TM} i7-2600 CPU with 4 physical cores and hyper-threading.

5.4 Different Logarithm Libraries

Additionally to using the `log` function from the standard C++ header `<math.h>`, we also used the `crlibm` library by Daramy et al. [**?**]. It includes versions of the natural logarithm computation, which allows the type of rounding to be explicitly specified.

In order to check the correctness of our resulting alignments, we computed the edit distance between the alignments we obtained by using different logarithm implementations. We assigned a cost of 1 to insertions, deletions and substitutions of single letters. We picked all pairs of sequences from the `BDNF_unaligned_sequences.fas` data set from `http://www.uni-oldenburg.de/fileadmin/user_upload/biologie/ag/systematik/download/Programs/benchMark_data.tar.gz`.

For the remaining parameters, we used $\lambda = 1, \mu = 2, \pi = (0.27, 0.24, 0.26, 0.23)$ and $\tau = 0.1$. The edit distances were computed in relation to using the logarithm function from `Boost.Multiprecision`.

In particular the `log_ru` function, which explicitly rounds up, produces a notable change in alignment compared to the logarithm function from `Boost.Multiprecision` as can be seen in Table 2. The computed likelihood scores were highly similar and the differences in the alignments were minimal

as can be seen in Figure 13. We finally decided to use the `log_ru` function from the `crlibm` library because it had the most consent with the alignments returned by the reference implementation (`http://sco.h-its.org/exelixis/web/teaching/practical15/scaledCode/tkf91_scaling.tar.gz`).

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

Table 2. Average edit distances from the alignment returned by using `Boost.Multiprecision` of different logarithm implementations.

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. Difference in alignments using the input parameters $\pi = (0.25, 0.25, 0.25, 0.25), \lambda = 1, \mu = 2, \tau = 0.1$

5.5 Runtime Measurements

The input sequences used to measure execution time were four pairs of randomly generated sequences of length 10, 100, 1000 and 10000 nucleotides. Sequences did not contain ambiguous characters.

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 output of the algorithm. Kernel execution was performed multiple times per input and subsequently averaged. The number of runs executed was dependent on the size of the input, so as to balance accuracy and overall benchmark time (see Table 3).

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

Table 3. Number of runs per sequence length

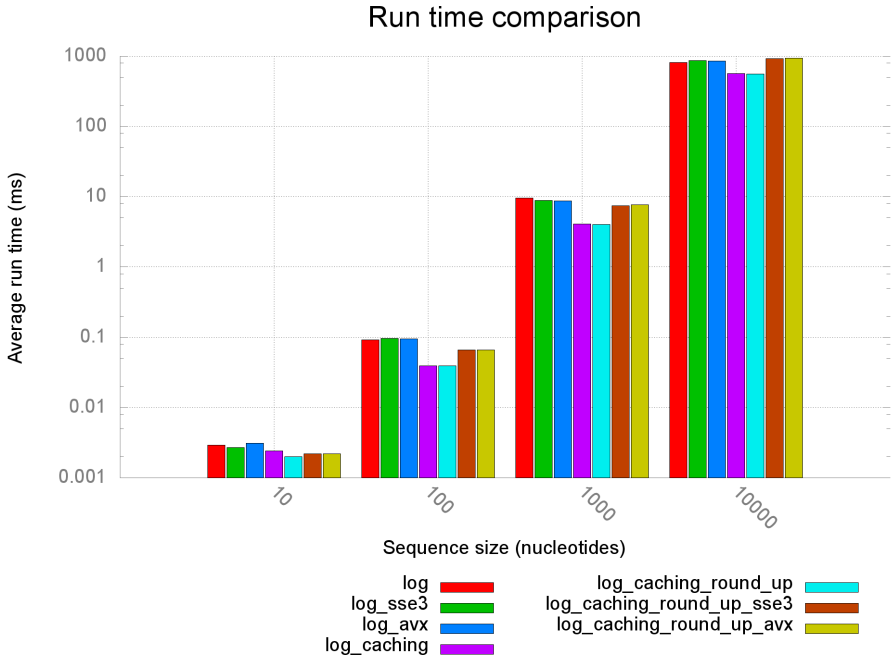

Fig. 14. Comparison of average run times of the different implementations. `sse3` and `avx` post-fixes denote the vectorized versions of the programs, using SSE3 and AVX vector intrinsics respectively (Section 4.3). `log` denotes, that a version built on the log-space transformed version of TKF91 (Section 4.1). Versions using the Array-of-Structs method of storing matrices (Section 4.2) are denoted by the `caching`-keyword, and the `round_up`-keyword denotes the use of 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.

TODO: apart from SIMD were there any other new techniques or tools (e.g. cache analysis, clang compiler etc.) you used for the first time

Well, we didn't do cache analysis and clang wasn't new to us either. We occasionally ran a profiler over our code, but I think that is covered with improving the code w.r.t. performance

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 `valgrind cachegrind` for the first time **correct?**. Finally, we concluded that `C++` is appropriate for HPC projects, provided that, the problem and data-structures at hand are well understood.

TODO: apart from SIMD were there any other new techniques or tools (e.g. cache analysis, clang compiler etc.) you used for the first time; SL: I didn't use clang before, only gcc. Perf was also new to me.

6.3 What did the Teacher learn?

TODO

7 Conclusion

For other courses, integration, even faster codes, some of the techniques to be used for more complex stat aligners such as ....