Generic accelerated sequence alignment in SeqAn using vectorization and multi-threading

Supplementary Material

René Rahn*[1], Stefan Budach[2], Pascal Costanza[3], Marcel Ehrhardt[1], Jonny Hancox[4], and Knut Reinert[1,2]

[1]Department of Mathematics and Computer Science, Freie Universität Berlin, Takustr. 9, 14195 Berlin, Germany
[2]Max Planck Institute for Molecular Genetics, Ihnestr. 63-73, 14195 Berlin, Germany
[3]imec, Belgium
[4]Intel Corporation (UK) Limited, United Kingdom

March 29, 2018

Contents

In this paper we studied a the acceleration of the prominent dynamic programming (DP) algorithms for pairwise sequence alignments. The foundation of this work was the unification of the different alignment algorithms resulting in a truly generic DP kernel that allowed us to accelerate a large class of algorithms by fully exploiting topical concurrency on modern hardware using multi-threading and vectorization.

In this supplementary document we will first describe the techniques and designs used to unify the DP kernel and subsequently how we used this to achieve generic vectorization and parallelization. In the end we will discuss the results of the benchmarks with respect to three different processor architectures.

1 The generic alignment module in SeqAn

The fundamental core of SeqAn's alignment module is built on a unified dynamic programming (DP) kernel. It is based on the central observation, that many of the aforementioned DP algorithms share the same execution profile. In particular, they only slightly differ in terms of their initialization, recursion, or *objective space*. As objective space we consider the cells in the matrix that need to be inspected to determine the correct result. For example, in case of a global alignment the objective space constitutes the last cell in the bottom right corner of the matrix while in the case of a local alignment all cells of the matrix form the objective space. Moreover, this profile is static during the execution of the DP algorithm, which gives us the opportunity to preconfigure the code at compile time. To do this, we devised a meta-model based on a column-wise execution flow over the DP matrix. The model is depicted in Fig. S1 for the non-banded and banded alignment case.

The DP matrix is divided into three column types: The `FirstCol`, `InnerCol`, and `LastCol` and each column is further split into three cell types: `FirstCell`, `InnerCell` and `LastCell`. In addition, we characterize different column spans, which can either be a `Full`-span covering the entire DP column or a `Top`-, `Middle`-, or `Bottom`-span for the different band positions, where the band crosses the top of the matrix, touches neither the top nor the bottom, or crosses the bottom of the matrix respectively. This conceptual partition is sufficient to formulate an unique execution profile for every cell in the DP matrix. This profile determines, how the corresponding cell is computed and whether it is part of the objective space. For example, the model in Fig. S1 for the non-banded case specifies a profile, where the first column and first row are initialized according to the standard global alignment initialization, whereas the global solution is searched in the entire last row and column, and hence generating a variant of the free-end gap global alignment. On the other hand, the model for the banded case depicts a local alignment computation. The different colors indicate different recursion dependencies for the respective cells. The dashed fields assemble the objective space in which the optimal solution is searched.

Each of the aforementioned column, cell, and span types are implemented as tags, which are stateless classes used to select different code paths (policies) at compile time. This strategy is also known as *tag-dispatching* utilized by the compiler to generate specific

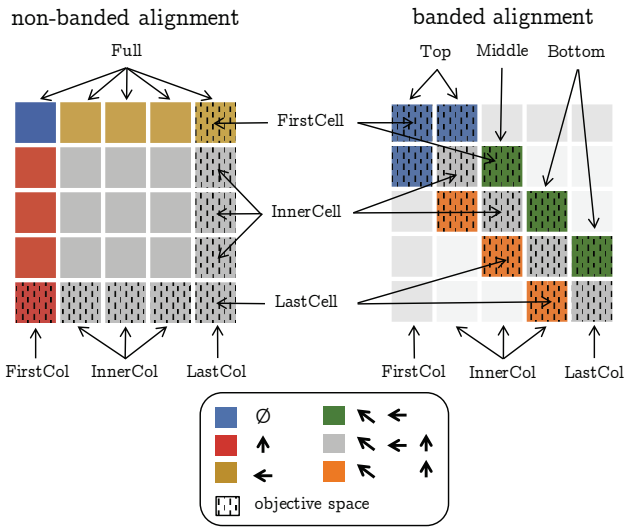

Figure S1: The conceptual partition of the DP matrix. We differentiate between three column types: `FirstCol`, `InnerCol` and `LastCol`. The same is done for dividing up each column into `FirstCell`, `InnerCell` and `LastCell`. Additionally, we use different column spans. In the non-banded case we compute `Full` columns, whereas in the banded case we differ between columns that intersect with the first row (`Top`), the last row (`Bottom`) or the columns in between, which can be either `Middle` or `Full` depending on the band settings. The legend shows the color encoded recursion dependencies for which we implemented specific compute kernels. The dashed fields indicate cells that are part of the objective space.

code, tailored to the execution profile of the respective alignment algorithm.

To reflect the meta-model in the execution of the DP matrix we unrolled the outer and the inner for loops of the DP kernel. Algorithm 1 shows the corresponding pseudocode. The function `compute_matrix` in Line 18 iterates over the columns of the matrix and calls for every column the function `compute_column` at Line 9 with the corresponding column span $\mathcal{S}$ and column type $\mathcal{C}$ as well as the current column index i. Here, $\mathcal{M}$ represents a state for all additional data necessary to compute the alignment, such as the scoring and trace matrix, the scoring function, a state for the current maximum, etc.. The function `compute_column` (Line 9) applies the same unrolling pattern by first computing the first cell, then all inner cells and finally the last cell. Every invocation of the function `compute_cell` (Line 1) uses the information of $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{Z}$ to a) select a recursion tag, to dispatch to the corresponding recursion kernel (see Fig. S1), and b) query whether the current cell is part of the objective space (Line 4). In the banded cases the implementation of the outer for-loop is slightly more complex, but follows the

Algorithm 1 Unified DP-core with unrolled loops

```
 1: function COMPUTE_CELL(i, j, s1, s2, M, S, C, Z)
 2:     Select recursion tag R based on S, C and Z.
 3:     compute(i, j, s1, s2, M, R)          ▷ tag-dispatching to recursion kernel.
 4:     if tracking is required then
 5:         check(M)
 6:     end if
 7: end function
 8:
 9: function COMPUTE_COLUMN(i, s1, s2, M, S, C)
10:     m ← |s2|
11:     compute_cell(i, 0, s1, s2, M, S, C, FirstCell{})
12:     for 1 < j < m do
13:         compute_cell(i, j, s1, s2, M, S, C, InnerCell{})
14:     end for
15:     compute_cell(i, m, s1, s2, M, S, C, LastCell{})
16: end function
17:
18: function COMPUTE_MATRIX(s1, s2, M)
19:     initialize M
20:     n ← |s1|
21:     compute_column(0, s1, s2, M, FullSpan{}, FirstCol{})
22:     for 1 < i < n do
23:         compute_column(i, s1, s2, M, FullSpan{}, InnerCol{})
24:     end for
25:     compute_column(n, s1, s2, M, FullSpan{}, LastCol{})
26:     return max of M
27: end function
```

same principle and reuses the `compute_column` function.

To configure the DP kernel we use a central *traits-class*. A traits-class, similar to tags, is also stateless but carries information used by the DP algorithm to determine the corresponding policies (implementation details). This class is a template class. At compile time the given types are evaluated and the corresponding policies making up the desired DP execution profile is configured.

1.1 Gap model

We offer three gap functions, the standard linear and affine gap model (Needleman and Wunsch, 1970; Gotoh, 1990) and an affine-like gap model (Urgese *et al.*, 2014), which speeds up the runtime, while at the same time providing results comparable to the affine gap model. Each gap function is realized as a separate policy, offering the most efficient implementation for the respective recursion. The function `compute` in Line 3 of Algorithm 1 is overloaded with the corresponding policy $\mathcal{R}$. In Section 2 we will explain how we modified these kernels to generically add vector-level parallelism to the DP kernel.

1.2 Traceback algorithm

SeqAn supports a large set of traceback options. At the moment we support five different traceback modes: *score_only*, *single_best*, *all_best* in combination with *left_gap* or *right_gap* projection. With the *score_only* option only the score is computed and reported but no alignment is generated. The default mode is the *single_best* mode, which choses one of the optimal tracebacks during the computation of the DP matrix. The *all_best* keeps track of all paths that result in an optimal alignment. We added an option that in the case of an ambiguous gaps placement the gaps are either projected to the left or to the right of the ambiguous gap field. Figure S2 depicts the different alignment outputs if ambiguous gap placement occurs. This distinction is used for the *split_breakpoint* implementation that we implemented in SeqAn for different applications Emde *et al.* (2012); Holtgrewe *et al.* (2015). In the *all_best* mode all possible optimal tracebacks are tracked, which is used in our banded-chain alignment algorithm Brudno *et al.* (2003) to compute an alignment along a set of given seed-anchors, while requiring only one alignment matrix to be kept in memory at a time.

We implemented a generic maximization kernel that reflects the different traceback settings and is reused by the kernels of the different gap functions. Again we use tag-dispatching to chose at compile time the correct maximization function.

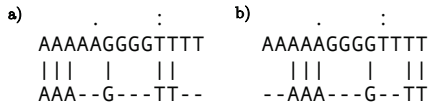

Figure S2: Comparison of a) right and b) left gap projection in case of ambiguous gaps placement.

1.3 Customization points

A major design goal of our implementation is the extendibility of the DP kernel, as many algorithms need to implement a specific behavior deviating from the standard DP algorithms. To achieve this we added a customization point that allows us to extend the DP core without invasive code adaptions or code duplications when adding new alignment policies. Moreover, features can be added that are orthogonal to other policies, such as gap or traceback policies, etc., so that these features are available even for new variants. To do this, we pass an algorithm-specific state into the DP kernel which can be queried and updated during the compute cell function in Line 1 of Algorithm 1. For example, the split-alignment algorithm requires the tracking of the column wide maximum for every column during the alignment computation and the x-drop implementation with affine gap costs terminates the alignment if the x-drop criteria is met (Hauswedell *et al.*, 2014). Both features are merely a specific implementation of this customization point.

2 Generic parallelization of the DP kernel

In this section, we will describe our generic vector- and thread-level parallelization for the alignment module, where we take advantage of the generic DP design accelerate a large set of alignment algorithms.

2.1 SIMD-abstraction

The difficulty for any library is to offer a broad set of target architectures it can be used on. This becomes even more prudent, if considering hardware specific optimization such as SIMD vectorization, as different architectures support different instruction sets. To generically add and support different SIMD-instruction sets we encapsulated the vector type as an alphabet type, which abstracts the vector width and the number of elements processed per vector. More precisely, a vector ν can store and process in parallel $l = \frac{|\nu|}{|\iota|}$ elements, with $|\nu| \in \{128, 256, 512\}$ being the width of the vector in bits and $|\iota| \in \{8, 16, 32, 64\}$ being the width of a single element (in our case an integer) in bits. In SeqAn we implemented SSE4 and AVX2 intrinsics and also added support for AVX512 intrinsics by using gcc's vector type extension and it's capabilities to auto-vectorize certain arithmetical and comparison expressions. To support a wider range of instruction sets we implemented a wrapper interface for the UME::SIMD vector library (Karpiński and McDonald, 2017) as an alternative SIMD-instruction backend. With UME::SIMD we can extend our vectorized alignment algorithms to target platforms using IMCI, AltiVec or NEON instructions. Furthermore, it extends our module to be compilable on Windows platforms.

2.2 Vector-level parallelization

Given this SIMD encapsulation, we can use the DP core as described in Algorithm 1 for sequences of vector-alphabets in addition to sequences with scalar-alphabets. In our

approach we generalized the inter-sequence vectorization layout, to calculate alignments in a many-to-many fashion, i.e., we allow multiple query sequences to be aligned against multiple subject sequences packed in one vector.

Let S_1 and S_2, with $|S_1| = |S_2|$ be two sets of sequences that are going to be aligned, such that $S_1[i]$ is aligned with $S_2[i]$, with $i \in [0, |S_1|)$. In the vectorized kernel, these sets are aligned in batches of size l. In total there are $\left\lceil \frac{|S_1|}{l} \right\rceil$ many batches to compute. The last batch might not be full, if $|S_1| \bmod l \neq 0$. To run the last batch vectorized as well, we fill it up with the last sequence from the set S_1, respectively S_2, until it reaches the size l. The artificially added sequences are than ignored in the final solution.

2.2.1 Sequence preparation

Before the vectorized alignment is computed, we transform the layout of the sequences in the batch from an array-of-structures (AoS) to structure-of-arrays (SoA) (see Fig. 2 in the corresponding manuscript). We generate a vector of SIMD-vectors, where the SIMD-vector at the i-th position contains the i-th characters of the l sequences contained in the batch, assuming that the l sequences have the same length. If the sequences differ in their lengths, we expand the smaller sequence by adding pad characters. For this case, we implemented a book-keeping mechanism that keeps track of the different end-points of the sequences. We adapted the loops of the `compute_matrix` and `compute_column` function of Algorithm 1 accordingly to transmit an early `LastCol`, respectively `LastCell` signal, to the compute kernel in case an end-point of at least one sequence is reached. Because the signaled end-point might not be valid for all pairs contained in the batch, we mask the elements in the vector that do not share an end-point with the current position within the DP matrix in either dimension, i.e., horizontal and vertical direction. Thus, the objective space is restricted to only the valid sequence pairs.

2.2.2 Adapting the recursion functions

The recursion function mainly consists of add, max and `compare` operations for the two sequence characters. The add-operator is implemented for the vector-alphabets such that it can be used interchangeably with scalar-alphabet types. We overloaded the max operation to reflect the different traceback options, for the score-only, single-trace, and all-trace mode, which have different complexities. In this way the most performant operations are selected for the respective traceback operation depending on the execution profile.

If a traceback was requested, we additionally store the horizontal and vertical coordinates for the traceback start in two vectors. Without loss of generality, we assume that the valid score width correlates with the length of the sequences. Hence, if the score width of 16 bit is sufficient to hold the alignment score without inducing an integer overflow/underflow, we require, that the lengths of the sequences do not exceed the maximal representable value of the used integer type. In this particular case the sequences cannot exceed the length of $2^{16} - 1$ bp. In a final step, the traceback is computed sequentially

for each of the l alignments in the current batch. Since following the traceback has linear runtime compared to the quadratic matrix computation the produced overhead for the sequential execution can be neglected.

2.2.3 Vectorized scoring scheme

The character comparison is implemented in a generic score class, which works interchangeably for scalar-alphabets and vector-alphabets. If the scoring matrix is simple in the sense that it only differentiates match from mismatch and different gap penalties, as it is often done for DNA based alignments (Pearson, 2013), we simply store the respective costs as vectors. When two alphabet types are compared, we use the efficient compare-and-blend operation to select the corresponding match and mismatch scores. Therefore, no lookup table needs to be generated.

In order to support more complex scoring matrices, like the BLOSUM or PAM-matrices for amino acid alphabets, we use gather instructions that were added with the *AVX* instruction set (Intel, 2016). A gather instruction reads values from a sparse vector and packs them into a SIMD-vector. These gather instructions are in general very expensive and thus removed from the inner loop by pre-calculating a score profile (Rognes, 2011). However, this is not possible in our many-to-many vectorization layout, as it would require to create a score profile for every possible combination of query characters in a SIMD-vector resulting in a $|\Sigma|^l$ large table for every position of the vectorized subject sequences, which would obliterate the purpose of this score profile.

Instead, we use the gather instructions to extract the corresponding scores from a scalar matrix. We added gather-wrapper instructions for 16 bit and 8 bit packed SIMD-vectors for the AVX instruction set, as only 32 bit and 64 bit packed vectors are supported. Additionally we added a fall-back loop, that sequentially fills the SIMD-vector in case the gather instruction is not supported. We leave it as future work to also implement a generalized profile based scoring scheme that can deal with complex scoring matrices more efficiently.

2.3 Thread-level parallelization

The wavefront model has been a successful strategy to parallelize a single pairwise alignment computation (Edmiston *et al.*, 1988; Liu *et al.*, 2001). In this model the cells along the minor diagonal of the DP matrix can be distributed to multiple threads. Computing each cell as a single entity on the different cores is a specialization of a blocked computation of the DP-matrix, with a block size of one. In our scheme, we use a block size β to split the matrix into disjunct execution units, called tiles. Every tile is mapped to exact one sub-alignment of the global DP matrix. Then, similar to Liu et. al (2014), we parallelize the alignment computation over the minor diagonal of these tiles. However, in our model we use a task queue, to represent ready-to-compute tiles, which are then popped dynamically by the threads, causing the execution to follow a dynamic wavefront along the minor diagonal rather then being strictly bound to it.

To do so, we construct a dependency graph which determines the dependencies for a

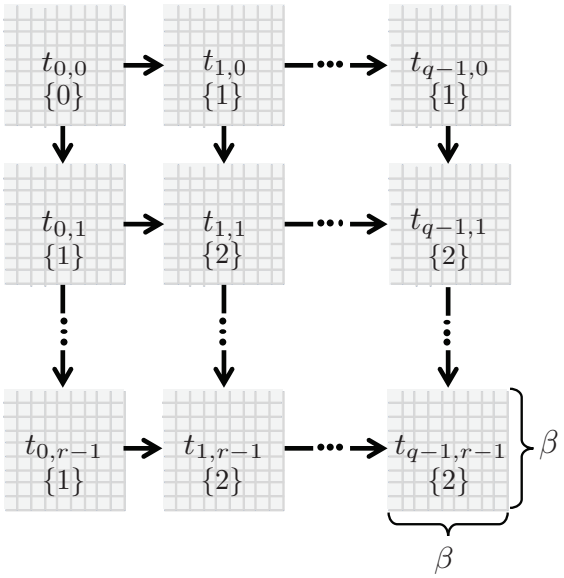

Figure S3: Dependency graph over a conceptual alignment matrix with associated tiles of size $\beta \times \beta$ and the corresponding dependency counts (in curly braces).

tile t. Fig. S3 depicts such a dependency graph. Except for the source and the tiles in the first column, respectively first row, all tiles have exactly two incoming edges. Hence, a tile $t_{i,j}$ can be executed if both it's predecessors $t_{i-1,j}$ and $t_{i,j-1}$ have been computed. In this case the dependency count of $t_{i,j}$ is decremented to 0 and is pushed onto the task queue. At some point one thread will pop this tile $t_{i,j}$ from the queue and execute

In order to compute a correct alignment it is necessary to synchronize the values of the last column and last row of each sub-alignment with their corresponding successors. This means that tile $t_{i+1,j}$ initializes the first column of its sub-alignment with the values of the last column of its predecessor $t_{i,j}$. Similarly, tile $t_{i,j+1}$ initializes the first row with the values from the last row computed in $t_{i,j}$ (see Fig. S4 green and blue buffer values). To avoid additional overhead through inter-thread communication we added a matrix wide horizontal buffer buf_h and vertical buffer buf_v, i.e., $buf_h[i]$ and $buf_v[j]$ store the corresponding initialization values, for the first row, respectively first column of tile $t_{i,j}$. These buffers are shared between the threads. To transmit the initialization values to the successor tiles we make use of our meta-model of the DP-matrix by implementing special `compute_cell` overloaded functions for the `FirstCol` and `FirstCell` tags, that instead of calling the standard compute kernel read their values from $buf_h[i]$ and $buf_v[j]$ respectively. Accordingly, we overloaded the `compute_cell` function for the `LastCol` and `LastCell` tags, where we overwrite the respective buffer values in $buf_h[i]$ and $buf_v[j]$ with the most recent DP values. The customization point is used to pass the corresponding buffers along with the currently executed sub-alignment. At the beginning of the alignment, the buffers are initialized for the selected alignment algorithm and gap function.

A noteworthy aspect of our model is the fact, that the dependency graph guarantees

that there is no race-condition on the buffer values, as it is not possible that two tiles in the same column or same row of the dependency graph are executed concurrently by different threads. Meaning, each worker thread can freely read from and write into the corresponding buffer without using expensive locks on the shared memory. Fig. S4 depicts a snapshot of a possible alignment wavefront with the use of the tile buffers buf_h and buf_v.

In addition, every thread has its own thread-local storage to keep track of the maximum score, which is only a local maximum over the sub-alignments computed by the respective thread. The matrix-wide global maximum is determined by the parent thread after the parallel execution finished computing the sink $t_{r-1,q-1}$, by a reduction over the stored maxima of each thread.

We wrapped the complex parallelized interfaces with SeqAn's known alignment interfaces, by adding a execution policy. With this execution policy the user can easily switch between three execution modes, namely: sequential, chunk and wavefront mode. All three of them can be combined with a vectorized execution. In the following listing, we show a sample code that demonstrates the efficiency of SeqAn's alignment interface using these execution policies.

```
#include <iostream>
#include <seqan/align_parallel.h>
```

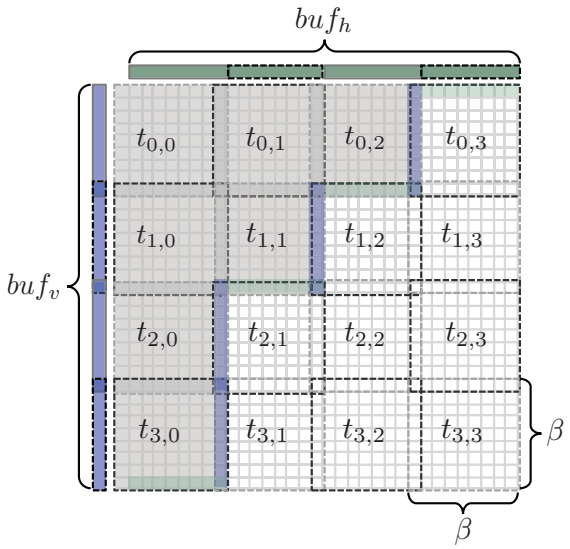

Figure S4: Schematic partitioning of the DP-matrix. Gray tiles are already computed or are currently processed by one thread. White tiles are not yet computed. Due to the work stealing mechanism, a dynamic wavefront progresses over the minor diagonal from the sink ($t_{0,0}$) to the source ($t_{3,3}$) of the corresponding dependency graph. The tiles overlap in one column and one row with their predecessor in the respective dimension. buf_h and buf_v are buffers used to synchronize the computed matrix values between tasks.

```
#include <seqan/stream.h>
#include <seqan/seq_io.h>

using namespace seqan;

template <typename ...TArgs>
auto invokeAlignment(TArgs &&... args)
{  // Dispatches to the corresponding alignment kernel.
    return globalAlignmentScore(args...);
}

int main(int argc, char const * argv[])
{
    StringSet<CharString> id1, id2;
    StringSet<DnaString> set1, set2;

    try {
        SeqFileIn seqFile1{argv[1]};
        SeqFileIn seqFile2{argv[2]};

        readRecords(id1, set1, seqFile1);
        readRecords(id2, set2, seqFile2);
    } catch (...) {
        std::cerr << "IOError or ParseError" << std::endl;
        return 1;
    }

    Score<int> sc{4, -5, -1, -11};
    String<int> res;

    if (atoi(argv[3]) == 1) {
        ExecutionPolicy<Parallel, Vectorial> execPol;
        res = invokeAlignment(execPol, set1, set2, sc);
    }
    else if (atoi(argv[3]) == 2) {
        ExecutionPolicy<WavefrontAlignment<>, Vectorial> execPol;
        setBlockSize(execPol, 2000);
        setParallelAlignments(execPol, 1000);
        res = invokeAlignment(execPol, set1, set2, sc);
    }
    else {  // Run without thread-level parallelization
        ExecutionPolicy<Serial, Vectorial> execPol;
        res = invokeAlignment(execPol, set1, set2, sc);
    }

    for (size_t i = 0; i < length(res); ++i) {
        std::cout << id1[i] << " vs " << id2[i] << " = " << res[i] << std::
            endl;
    }
    return 0;
}
```

Listing S1: Example code demonstrating the efficiency of the execution policies. Depending on the user input (third argument to the program) either of the three execution policies: *chunked*, *wavefront*, or *sequenced* can be invoked. All execution policies can also be combined with the the tag `Serial` in the second template argument to disable vectorized execution.

3 Evaluation of the generic accelerated alignment module.

To study the effectiveness of our generic accelerated alignment module we used three different microprocessor architectures. The first processor is a 2 socket Intel® Xeon® E5-2650 V3 CPU (Haswell) with a total of 20 physical cores (10 per socket) at a base frequency of 2.3 GHz. In the following we refer to this CPU as *HSW*. The second CPU, in the following referenced as *SKX*, is a 2 socket Intel® Xeon® Gold 6148 CPU (Skylake) with 40 cores (20 per socket) at a base frequency of 2.4 GHz. In addition, we run our experiments on an Intel® Xeon Phi™ 7250 CPU (Knights Landing) with 68 cores at a base frequency of 1.4 GHz provided by the HLRN test and development system hosted at the Zuse Institute Berlin. The Xeon Phi™, in the following referred to as *KNL*, was booted into the cache quadrant mode, which utilizes the associated on-chip high-bandwidth memory, also known as MCDRAM, as an additional cache to the main memory.

All systems were running a linux operating system. The HSW is an older processor architecture supporting SIMD instructions up to AVX2, with 16 256 bit registers per core. The SKX and KNL processor both support AVX512 with 32 512 bit registers, but in addition SKX implements AVX512-BW, which adds byte (8 bit) and word (16 bit) instructions for the 512 bit registers, which is not available on the KNL. On all platforms we used g++-7.2.0 to compile the binaries for the evaluation. The SeqAn benchmark binaries were compiled in release mode with `-std=c++14 -fopenmp -mpopcnt -O3 -DNDEBUG`. The following additional flags were used on HSW: `-mavx2`; on SKX: `-mavx512f -mavx512cd -mavx512bw -mavx512dq -mavx512vl`; and on KNL: `-mavx512f -mavx512cd -mavx512er -mavx512pf`. All other tools have been compiled with optimizations turned on (`-O3 -NDEBUG`) and with the corresponding instruction subsets according to the accompanied build scripts. For all CPUs we disabled the Intel®turbo boost feature and fixed the cores to their respective base frequency.

3.1 Vectorization

To test our generalized inter-sequence vectorization model we run the first use case, aligning Illumina reads, with our `align_bench_chunk` tool. Table S1 presents the data for all three CPUs executing with all available instruction sets and with the maximum number of physical cores available on the corresponding system. Each sub-column presents the used instruction set. We tested our module in the non-banded and banded mode, where

			HSW (t = 20)		SKX (t = 40)			KNL (t = 68)		
			sse4_16	avx2_16	sse4_16	avx2_16	avx512_16	sse4_16	avx2_16	avx512_32
seqan	global	time (s)	5.90	2.80	2.68	1.35	0.68	10.81	4.11	2.35
		GCUPS	48.28	101.90	106.27	211.32	420.41	26.36	69.33	121.45
		Factor	1.51	3.19	3.31	6.59	13.11	0.82	2.16	3.79
	semi	time (s)	5.90	2.76	2.68	1.35	0.68	10.81	4.11	2.34
		GCUPS	48.27	103.08	106.26	211.47	420.42	26.36	69.35	121.73
		Factor	1.43	3.06	3.34	6.64	13.20	0.83	2.18	3.82
	local	time (s)	7.38	3.48	3.26	1.55	0.80	14.96	6.35	2.53
		GCUPS	38.60	81.86	87.43	183.76	354.44	19.05	44.84	112.55
		Factor	1.13	2.40	2.66	5.58	10.77	0.58	1.36	3.42
seqan_band	global	time (s)	1.39	0.85	0.62	0.40	0.24	1.64	0.93	0.75
		GCUPS	23.13	37.85	51.33	79.46	135.53	19.52	34.48	42.98
		Factor	6.44	10.54	14.22	22.02	37.55	5.41	9.55	11.91
	semi	time (s)	1.36	0.85	0.61	0.43	0.25	1.64	0.97	0.77
		GCUPS	23.54	37.81	52.54	74.57	128.07	19.55	32.91	41.58
		Factor	6.20	9.96	14.65	20.80	35.72	5.45	9.18	11.60
	local	time (s)	1.47	0.91	0.74	0.42	0.26	2.09	1.07	0.84
		GCUPS	21.76	35.32	43.27	76.34	125.13	15.33	29.96	38.04
		Factor	5.66	9.18	11.67	20.60	33.76	4.14	8.08	10.26
parasail	global	time (s)	8.93	9.40	3.88	4.00		11.67	13.53	
		GCUPS	31.91	30.33	73.40	71.21	—	24.41	21.06	—
		Factor	1.00	0.95	2.17	2.10		0.77	0.66	
	semi	time (s)	8.45	8.87	3.66	3.88		11.64	13.75	
		GCUPS	33.71	32.11	77.86	73.43	—	24.48	20.73	—
		Factor	1.00	0.95	2.16	2.17		0.77	0.65	
	local	time (s)	8.34	8.12	4.11	3.96		12.19	15.39	
		GCUPS	34.17	35.11	69.41	71.68	—	23.38	18.51	—
		Factor	1.00	1.03	1.99	2.14		0.71	0.56	

Table S1: Performance evaluation of the vectorized DP kernel for the global, local and semi-global alignment using the non-banded and banded version of SeqAn and comparing them to Parasail on all three test platforms: HSW, SKX, and KNL. The benchmarks were executed on each platform using OpenMP with the number of threads set to the respective number of physical cores available. For every alignment case we choose Parasails runtime on the HSW with SSE4 as the base line for the calculated factor.

the band size was 8 bp in each direction to represent an error rate of 5%. We also compared against Parasail, were we chose the configuration that produced the best results. Each row is sub-divided into global, semi and local alignment for which we provide the runtime in seconds as well as the achieved GCUPS (Giga Cell Updates Per Second). For convenience, we added a factor to compare the runtimes for global, semi-global and local alignment between the different systems and the different instruction sets. We chose the runtime achieved with Parasail on the HSW system executed with SSE4 as the baseline for the respective alignment mode (factor 1).

At first, we compare our results with the Parasail library, and afterwards evaluate the results with respect to the different architectures. In general, SeqAn's generalized inter-sequence vectorization scheme outperforms Parasail in all but one case, which is the local alignment for SSE4 on the KNL (factor 0.58 vs. 0.71). Furthermore, our scheme scales perfectly with larger register sizes, while Parasail seems to be limited for this kind of data, as the runtime decreases poorly and sometimes even increases on

AVX2. AVX512 is currently not supported by Parasail, so that we could not compare to it. The banded alignment allows great speedups, albeit it's relative speed, measured in GCUPS, decreases compared to the non-banded implementation. In the banded matrix the work reduces from a quadratic to a linear problem scaled with a small constant. Thus, the total overhead induced by the parallel execution (vectorization and multi-threading) contributes significantly more to the total runtime affecting the relative speedup. Nonetheless, our banded version outperforms the non-banded case by a factor of about three on all platforms. On the SKX we could achieve a speedup factor of 37 for the global alignment, which is 17 times faster than the fastest configuration of Parasail on the SKX.

Comparing the results between HSW and SKX emphasizes the stability of our parallelization on machines with higher number of physical cores and modernized microprocessor architectures. On HSW we measured slightly lower runtimes for AVX2, than on SKX with SSE4 (e.g., 2.8 seconds for global alignment on HSW with AVX2 as opposed to 2.68 seconds on SKX with SSE4), which we would expect, as the number of alignments per vector halves while at the same time the number of cores doubles. Given the slightly higher clock speed of the SKX the results mirror perfectly the system's specifications. On SKX we could make use of the AVX512-BW instruction feature, which adds byte and word instructions for the 512 bit registers. Again we see a perfect scaling compared to AVX2 on SKX for the non-banded case. The banded alignments scale only with a factor of roughly 1.5 to 1.7, which can also be explained by the proportional higher overhead of converting the sequences into their vector representation.

The best performance on the KNL using AVX512 with 32 bit packed integers is slightly better than the best performance measured on the HSW using AVX2 with 16 bit packed integers. In both cases 16 alignments are computed in parallel in one vector. Moreover, both SSE4 and AVX2 do not perform well on the KNL. The AVX2 implementation is by a factor from 1.75 to 2.5 slower than the AVX512 implementation for the non-banded implementation, although one would expect that they would roughly yield the same result. But since there is no native support for 8 bit and 16 bit instructions on the KNL, these instructions might be executed with much higher latency than natively supported instructions. Compared to the SKX, the KNL is slower by a factor from 3 to 3.5. However, on SKX twice as many alignments can be computed in the vector unit due to the AVX512-BW instruction set and the clock rate is much higher.

Comparison to sequential scalar execution: We also compared the efficiency of the modernized DP Kernel regrading the vectorization by comparing against the previous sequential scalar DP kernel of SeqAn's alignment module. In Table S2 we show the gained speedup when executing different number of alignments for different sequence lengths. The experiments were done with our benchmark app, which also has a sequential mode without multi-threading but can switch between the vectorized kernel and the previous scalar kernel via a command line argument.

Considering a single alignment computation we expected that there is potentially no benefit over the scalar execution as the alignment cannot be parallelized in our model.

#alignments	100bp	1000bp	2000bp	5000bp
1	1.01	1.86	1.52	0.65
16	13.25	28.67	27.68	10.93
32	14.03	31.74	31.58	10.77
64	19.12	32.08	32.30	10.80

Table S2: Speedup of the vectorized DP kernel over the previous scalar version for different number of alignments and different sequence lengths. The benchmarks were executed single threaded on the KNL with AVX512 using 32 bit scores (16 packed alignments).

Surprisingly, for smaller sequences up to medium sized sequences of 1Kbp to less than 5Kbp, the vectorized kernel shows a super-linear speedup despite the overhead of transforming the sequence into it's SIMD representation. In particular, the vectorized kernel for a single alignment computation with two sequences of length 1Kbp is 1.86 times faster than the previous scalar kernel of SeqAn. When computing multiple alignments we even gain a 32 fold speedup.

This super-linear speedup can be explained by the following observations. First, the scalar kernel might not be as optimized as the vectorized kernel, as the character comparison and the score comparisons in every cell are done using either an if-clause or the ternary operator, which potentially induce conditional jumps. In contrast, the vectorized kernel implements these as compare-and-blend operations, such that no conditional jumps are performed and the compiler might be able to optimize the corresponding instructions more efficiently. Second, the cache structure of the underlying processor might influence the performance depending on the sequence lengths. In case of the KNL, two cores share a L2 cache. Since both cores do not share any data during the alignment computation, the available cache size per core is presumably halved. The packed sequences and the vectorized DP matrix require more space than the scalar kernel, since there needs to be 16 times more data stored in the cache per execution cycle. If the sequences are too small, the overhead of transforming the data into SIMD registers is to expensive. If the sequences become too long, the data of an entire DP column won't fit into the cache and access to the next cache level is required, which adds another performance penalty. But for sequence lengths of about 2Kbp the cache size seems to be large enough to keep the data close to the core, such that the presumably better optimized SIMD kernel achieves better performance than the scalar kernel. On the SKX using thirty two 16 bit packed vector units, we could measure a speedup of 1.3447 for the computation of a single alignment with 3Kbp long sequences over the scalar version. When executing 32 alignments the speedup was 45, which is also a super-linear speedup. The larger sequence sizes can be explained by a bigger cache size.

3.2 Wavefront parallelization

In the next section we evaluate the scalability of the dynamic wavefront implementation for our multi-threaded intra-sequence parallelization without vectorization. We aligned Enterobacteria phage SPC35 (Id: NC_015269.1; length: 118351 bp) with Salmonella

phage Shivani (Id: NC_028754.1; length: 120098 bp) and Chlamydia trachomatis D/UW-3/CX chromosome (Id: NC_000117.1; length: 1042519 bp) with Thermotoga maritima MSB8 chromosome (Id: NC_000853.1, length: 1860725 bp) on the KNL.

Fig. S5 shows the scalability of our wavefront implementation for both alignments for different block sizes up to 68 threads. The thick dashed blue line represents the optimal scaling curve based on the sequential scalar execution of the alignment over the number of threads. For the smaller alignment the wavefront implementation scales perfectly until 16 threads, independent of the block size (see Fig. S5a). Thereafter, it shows, that the larger the block size is the less parallelism can be exploited and the the performance gain saturates. However, with a block size of 500 the scaling remains still optimal. On the other hand, Fig. S5b shows, that with longer sequences the scaling becomes optimal even for larger block sizes, which are beneficial for the vectorization, as the proportional overhead for converting the sequences becomes smaller. Thus, with longer sequences a bigger block size will still produce enough work for the scheduler as there are enough blocks along the minor diagonal to keep all cores busy. This changes with smaller sequences, where also the block size needs to be adapted to produce enough work.

	SKX (t = 40; avx512_16)				KNL (t = 68; avx512_32)			
	β	time	GCUPS	Factor	β	time	GCUPS	Factor
D4.4M vs. D4.6M	2000	137.61	148.81	816.87	1100	327.57	62.51	1271.11
D23M vs. D33M	2500	3155.64	239.18	1312.96	1900	9487.54	79.55	1617.62
D23M vs. D42M	3000	3831.22	252.47	1385.95	1900	12009.70	80.54	1637.73
D23M vs. D50M	3000	4601.75	250.40	1374.56	1900	14605.50	78.89	1604.21
D33M vs. D42M	3000	5327.20	258.80	1420.70	1900	17477.40	78.88	1604.04
D33M vs. D50M	3000	6384.45	257.25	1412.15	1900	20374.70	80.61	1639.09
D42M vs. D50M	3000	8147.52	258.34	1418.16	1900	26426.00	79.65	1619.60

Table S3: Performance evaluation of the vectorized wavefront alignment for single large-scale sequence alignments on the SKX and the KNL. The factor column is based on the GCUPS sampled for the sequential scalar algorithm on the respective platform.

With the large-scale sequence alignment we evaluate the performance of the combined acceleration using the wavefront thread-level parallelization together with the inter-sequence vectorization for a single alignment. We performed this benchmark on the SKX and the KNL. The corresponding data can be reviewed in Table S3. Again if comparing both architectures, it shows that SKX is by a factor of 3 faster than the KNL. The main difference is due to the 512-BW instruction set which is not supported by the KNL. Thus we could only use the 32-bit packed vector instructions on the KNL. Second, the the cores of the SKX processor run with a much higher clock rate, which explains the time difference. However, if comparing to a scalar non-vectorized execution of the same alignments on the respective processor it shows that the relative speedup on the KNL is much higher reaching a total performance increase of the algorithm by a factor of roughly 1600. The super-linear speedup and the optimal block-sizes strongly correspond to the observations we made in 3.1.

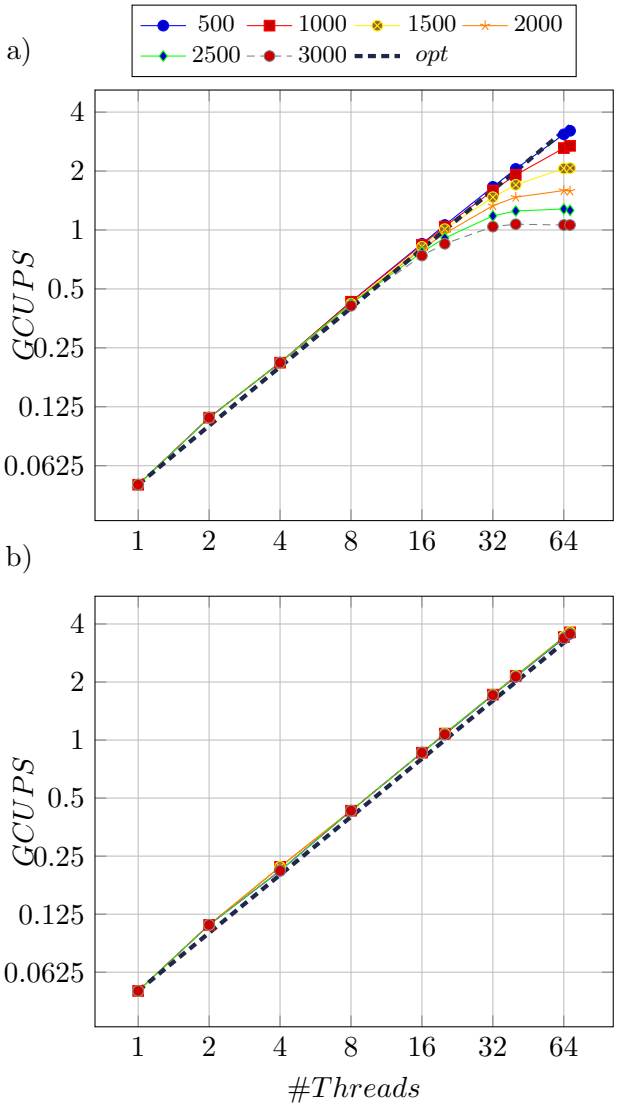

Figure S5: In a) is shown the results for the small sequences of $\sim$ 120 Kbp length and in b) for the longer sequences larger than 1 Mbp.

We tried to compare our algorithm against *SWAPHI-LS* from Liu *et al.*, but the application is only available for the predecessor (KNC coprocessor) of the Xeon Phi™ as an offload application. Although, we were not able to compile it for the KNL, in their paper they offloaded the same alignments to multiple Xeon Phis™ , where they obtained a peak performance on 2 Xeon Phis™ of about 55 GCUPS, which is still 25 GCUPS slower than our model on a single Xeon Phi™.

In the last benchmark we measured the performance of the generalized alignment scheduler, which allows to run multiple alignments concurrently with our wavefront implementation.

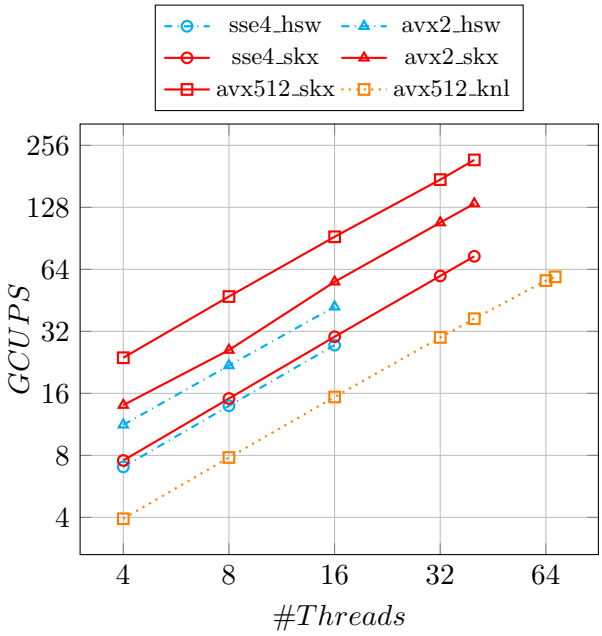

Figure S6: Performance and scalability comparison of aligning the PacBio-Sim data on HSW, SKX and KNL.

Fig. S6 shows the results for the simulated PacBio data on all three test platforms. The results underline the observations we have made with the pure vectorization evaluation using the Illumina test data set. In general HSW is slightly slower in the performance than SKX with the same instruction set and the same number of threads. AVX512 greatly improves the performance, albeit the scaling is with a factor 1.7 not as optimal as observed in Table S1. Furthermore, the KNL despite, it's many cores, can only slightly improve the runtime in the peak over the HSW. The measured peak performance was 217 GCUPS on the SKX, 59 GCUPS on the KNL and 53 on the HSW. This is in general lower than the peak performance measured for the large-scale sequence alignments, which can be explained by the more homogenous data of the long sequences. In this case most of the alignment tiles have the same length such that additional bookkeeping for different lengths of the sequences in the vectorized kernel can be omitted. The same observation can be made for the AVX512 results. Since twice as many alignments can be computed in one vector the more likely it is that the tiles executed simultaneously differ in their lengths resulting in an additional overhead for the bookkeeping. Hence, the less heterogeneous the sequence lengths in the data sets are the more efficient is the vectorization. We will optimize the vectorization approach for sequences with different lengths in order to achieve a more stable performance growth independent of the composition of the data sets.

We also compared our implementation against the *ksw2-test* tool from the *ksw2-*

library[1], which is used in minimap2 (Li, 2017) for the alignment computation. Unfortunately, the test tool is only single threaded, but we executed it on the HSW to compare it against our parallelization strategy without multi-threading. In our benchmark ksw2 achieved 1.56 GCUPS while SeqAn achieved 1.85 GCUPS using SSE4, respectively 2.83 GCUPS using AVX2, to align the PacBio-Sim data set, which is 1.2 to 1.8 times faster than ksw2.

[1] `https://github.com/lh3/ksw2`, accessed 21th March 2018

References

Brudno, M., *et al.* (2003). LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA. *Genome Res.*, **13**(4), 721–31.

Edmiston, E. W., *et al.* (1988). Parallel processing of biological sequence comparison algorithms. *Int. J. Parallel Program.*, **17**(3), 259–275.

Emde, A. K., *et al.* (2012). Detecting genomic indel variants with exact breakpoints in single- and paired-end sequencing data using splazers. *Bioinformatics*, **28**(5), 619–627.

Gotoh, O. (1990). Optimal sequence alignment allowing for long gaps. *Bull. Math. Biol.*, **52**(3), 359–373.

Hauswedell, H., *et al.* (2014). Lambda: The local aligner for massive biological data. In *Bioinformatics*, volume 30.

Holtgrewe, M., *et al.* (2015). Methods for the detection and assembly of novel sequence in high-throughput sequencing data. *Bioinformatics*, **31**(12), 1904–1912.

Intel (2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

Karpiński, P. and McDonald, J. (2017). A high-performance portable abstract interface for explicit SIMD vectorization. *Proc. 8th Int. Work. Program. Model. Appl. Multicores Manycores - PMAM'17*, pages 21–28.

Li, H. (2017). Minimap2: versatile pairwise alignment for nucleotide sequences. *arXiv preprint arXiv:1708.01492*, pages 1–6.

Liu, X., *et al.* (2001). Pacific Symposium on Biocomputing 6:127-138 (2001). *Symp. A Q. J. Mod. Foreign Lit.*, **138**, 127–138.

Liu, Y., *et al.* (2014). SWAPHI-LS: Smith-Waterman Algorithm on Xeon Phi coprocessors for Long DNA Sequences. In *2014 IEEE Int. Conf. Clust. Comput. Clust. 2014*, pages 257–265.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similiarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**(3), 443–453.

Pearson, W. R. (2013). Selecting the right similarity-scoring matrix. *Current protocols in bioinformatics*, pages 3–5.

Rognes, T. (2011). Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, **12**(1), 221.

Urgese, G., *et al.* (2014). Dynamic Gap Selector: A Smith Waterman Sequence Alignment Algorithm with Affine Gap Model Optimisation. *Proc. IWBBIO*, pages 1347–1358.