IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, IL, USA, May 2016.

AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-based Multi- and Many-core Processors

Kaixi Hou, Hao Wang, and Wu-chun Feng
Department of Computer Science
Virginia Tech, Blacksburg, USA
{*kaixihou, hwang121, wfeng*}*@vt.edu*

***Abstract*—Pairwise sequence alignment algorithms, e.g., Smith-Waterman and Needleman-Wunsch, with adjustable gap penalty systems are widely used in bioinformatics. The strong data dependencies in these algorithms, however, prevents compilers from effectively auto-vectorizing them. When programmers manually vectorize them on multi- and many-core processors, two vectorizing strategies are usually considered, both of which initially ignore data dependencies and then appropriately correct in a subsequent stage: (1) *iterate*, which vectorizes and then compensates the scoring results with multiple rounds of corrections and (2) *scan*, which vectorizes and then corrects the scoring results primarily via one round of parallel scan. However, manually writing such vectorizing code efficiently is non-trivial, even for experts, and the code may not be portable across ISAs. In addition, even highly vectorized and optimized codes may not achieve optimal performance because selecting the best vectorizing strategy depends on the algorithms, configurations (gap systems), and input sequences.**

Therefore, we propose a framework called AAlign to *automatically* vectorize pairwise sequence alignment algorithms across ISAs. AAlign ingests a sequential code (which follows our generalized paradigm for pairwise sequence alignment) and automatically generates efficient vector code for *iterate* and *scan*. To reap the benefits of both vectorization strategies, we propose a hybrid mechanism where AAlign automatically selects the best vectorizing strategy at runtime no matter which algorithms, configurations, and input sequences are specified. On Intel Haswell and MIC, the generated codes for Smith-Waterman and Needleman-Wunsch achieve up to a 26-fold speedup over their sequential counterparts. Compared to the highly optimized and multi-threaded sequence alignment tools, e.g., SWPS3 and SWAPHI, our codes can deliver up to 2.5-fold and 1.6-fold speedups, respectively.

***Keywords*-parallelization; vectorization; SIMD; automated code generation; alignment; pairwise sequence search; multicore; many-core; framework**

I. Introduction

Pairwise sequence alignment algorithms, e.g., Smith-Waterman (SW) [1] and Needleman-Wunsch (NW) [2], are important computing kernels in bioinformatics applications ([3], [4], [5]) to quantify the similarity between pairs of DNA, RNA, or protein sequences. This similarity is captured by a matching score, which indicates the minimum number of deletion, insertion, or substitution operations with penalty or award values to transform one sequence to another. To boost their performance on modern multi- and many-core processors, it is crucial to utilize the vector processing units (VPUs), which support single-instruction, multiple-data (SIMD) operations. However, the strong data dependencies among neighboring cells prevent such algorithms from taking advantage of compiler auto-vectorization. Thus, programmers need to explicitly vectorize their code or even resort to writing assembly code to attain better performance.

The manual vectorization of such algorithms often relies on two potential strategies: (1) *iterate* [6], [4], [5], which partially ignores the dependencies in one direction, vectorizes the computations, and then may compensate the results by using multiple rounds of corrections; or (2) *scan* [7], which completely ignores the dependencies in one direction, vectorizes the computations, and re-calculates the results with "weighted scan" operations and another round of correction. Either strategy has its own benefits depending on selected algorithms (e.g., SW or NW), gap systems (linear or affine), and input sequences.

Two main challenges face programmers. First, manual vectorization requires substantial coding effort in order to handle idiosyncratic vector instructions. For applications having complex data dependencies, expert knowledge of vector instruction sets and proficient skills to organize vector instructions are necessary to achieve the desired functionality. Moreover, current vector ISAs evolve quickly and some versions are *not* backwards-compatible [8]. Porting existing vectorized codes to other platforms becomes a laborious and tedious task. Second, even highly optimized vector codes may not achieve optimal performance at the application level. For pairwise sequence alignment, the combination of algorithms, vectorization strategies, configurations (gap penalty systems), and input sequences at runtime may lead to significant variability in performance. Furthermore, it increases the complexity of optimizing applications on modern multi- and many-core processors. Therefore, looking for a way to get around these obstacles is of great importance.

In this paper, we propose a framework called AAlign to automatically vectorize pairwise sequence alignment algorithms across ISAs. Our framework takes sequential algorithms, which need to follow our generalized paradigm for pairwise sequence alignment, as the input and generates vectorized computing kernels as the output by using the formalized vector code constructs and linking to the platform-specific vector primitives. Two vectorizing strategies are formalized as ”striped-iterate” and ”striped-scan” in our framework. In addition, we propose a hybrid approach that automatically switches between striped-iterate and striped-scan vectorization based on the context at runtime, which in turn provides better performance than past approaches.

The major contributions of our work are three-fold. First, we propose the AAlign framework, which can automatically generate parallel codes for pairwise sequence alignment via a combination of algorithms, vectorizing strategies, and configurations. Second, we show that the existing vectorizing strategies cannot always deliver optimal performance even when the codes are highly vectorized and optimized. As a result, we design a hybrid approach that takes advantage of two vectorizing strategies: iterate and scan. Third, using AAlign, we generate various parallel codes for the different combinations of algorithms (SW and NW), vectorizing strategies (striped-iterate, striped-scan, and hybrid), and configurations (linear and affine gap penalty systems) on two x86-based platforms, i.e., Advanced Vector eXtension (AVX2)-supported multicore and Initial Many Core Instruction (IMCI)-supported manycore.

We conduct a series of evaluations of the generated vector codes. Compared to the optimized sequential codes on the Haswell CPU, our codes using striped-scan vectorization can deliver 4.0-fold to 6.2-fold speedup, while using striped-iterate vectorization, our codes can provide 4.7-fold to 10.0-fold speedup. The vectorized codes also show performance benefit on Intel MIC and can achieve 9.1-fold to 16.0-fold speedup using striped-scan and 9.5-fold to 25.9-fold speedup using striped-iterate over the optimized sequential counterparts, respectively. We also compare our proposed hybrid approach with striped-iterate and striped-scan vectorization and show that our hybrid approach can achieve better performance on both platforms. Next, after wrapping our vector codes with multi-threading, we compare our hybrid approach to highly optimized sequence alignment tools, i.e., SWPS3 [4] on CPU and SWAPHI [5] on MIC. When aligning query sequences to an entire database, our codes achieve up to a 2.5-fold speedup over SWPS3 on CPU and 1.6-fold speedup over SWAPHI on MIC.

II. Background

This section provides a brief overview of (1) the vector ISAs of x86-based processors for both CPU and MIC and (2) the pairwise sequence alignment algorithms.

A. Vector ISA

Modern x86-based processors (e.g., CPU and MIC) are equipped with vector processing units (VPUs). These VPUs can carry out a single operation over a set of data simultaneously. Alongside, the vector ISA provides an abundant set of instructions and continues to evolve and expand to provide additional functionality, e.g., Advanced Vector Extensions (AVX), Initial Many Core Instructions (IMCI), and the incoming AVX-512 [9]. Meanwhile, the vector width has also extended from 128 bits (4 floats) to 256 bits (8 floats) to current 512 bits (16 floats), improving the throughput of systems and offering potential benefits for applications. In this paper, we focus on AVX2 and IMCI.

AVX2 Instructions: The width of AVX2 registers is 256 bits, consisting of two 128-bit lanes. The ISA is available since the Haswell architecture. AVX2 expands most vector integer SSE and AVX instructions to 256 bits and supports variable-length integers. In addition, AVX2 introduces gather, cross-lane permute and per-element shift instructions.

IMCI Instructions: The width of IMCI registers is 512 bits in four 128-bit lanes. IMCI works on the Knights Corner MIC architecture. Although IMCI provides additional functionality, e.g., scatter, gather, reduce, etc., it does not support previous vector ISAs, e.g., SSE and AVX. Because IMCI does not support 8-bit or 16-bit integers, we only consider 32-bit integers on IMCI in this paper.

B. Pairwise Sequence Alignment Algorithms

Algorithms for pairwise sequence alignment aim to quantify the best-matching score between two input sequences of DNA, RNA, or proteins. Specifically, the alignment uses the edit distance to describe how to transform one sequence into another by using a minimum number of predefined operations, including insertion, deletion, and substitution, along with an associated penalty or award. A common technique leverages dynamic programming, which uses tabular computations as shown in Fig. 1. If the input sequences are query Q_m with m characters and subject S_n with n characters, we need a $m * n$ table T, and every cell $T_{i,j}$ in the table stores the optimal score of matching the substring Q_i and S_j. To assist in the computation, we define three additional tables: $L_{i,j}$, $U_{i,j}$, $D_{i,j}$ denoting the optimal scores of matching with substring Q_i and S_j but ending with the insertion, deletion, and substitution, respectively. We can derive:

$$T_{i,j} = max(L_{i,j}, U_{i,j}, D_{i,j}) \tag{1}$$

Fig. 1 also shows the data dependencies. Visually, $L_{i,j}$, $U_{i,j}$, $D_{i,j}$ rely on its left, upper, diagonal neighbors. Although the algorithm takes $O(m * n)$ time and space complexity, we use a double-buffering technique, as shown by the two solid rectangles in the figure, to address the former while lowering the space complexity to $O(m)$, assuming the computation goes along the Q_m.

There are two major classes of pairwise sequence alignment algorithms: local alignment and global alignment. For the former, the Smith-Waterman (SW) algorithm [1] can quantify the optimal match with respect to partial (local) regions. For the latter, the Needleman-Wunsch (NW) algorithm [2] can find the best-matching score over entire sequences. Both algorithms have multiple variants by using

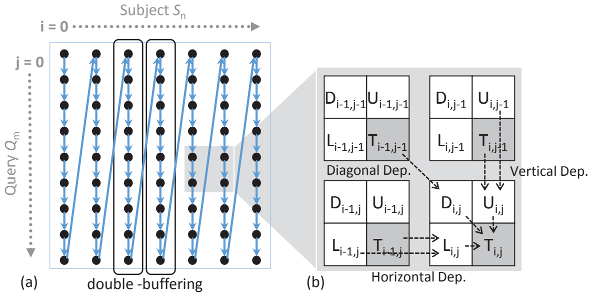


Figure 1: Data dependencies in the alignment algorithms using dynamic programming

linear or affine gap penalties. In Sec. IV, we show a generalized paradigm for pairwise sequence alignment algorithms.

III. Challenges

Alg. 1 shows the sequential code of SW with an affine gap penalty. Though writing the sequential code is relatively simple, vectorizing such an algorithm is non-trivial due to the strong data dependencies among the neighbors shown in Fig. 1.

Algorithm 1: Sequential Smith-Waterman following the paradigm (Sec. IV)

```
/* GAP_OPEN and GAP_EXT are constants; BLOSUM62 is a
   substitution matrix; ctoi is a user-defined function
   to map given character to the index number in the
   substitution matrix                                  */
for $i$ ←0; $i < n$+1; $i$++ do
    $T_{0,i} = U_{0,i} = L_{0,i} = 0$;
for $j$ ←0; $j < m$+1; $j$++ do
    $T_{j,0} = U_{j,0} = L_{j,0} = 0$;
for $i$ ←1; $i < n$+1; $i$++ do
    for $j$ ←1; $j < m$+1; $j$++ do
        $L_{i,j} = max(L_{i-1,j} + GAP_{EXT}, T_{i-1,j} + GAP_{OPEN})$;
        $U_{i,j} = max(U_{i,j-1} + GAP_{EXT}, T_{i,j-1} + GAP_{OPEN})$;
        $D_{i,j} = T_{i-1,j-1} + BLOSUM62_{ctoi(Q_{j-1}),ctoi(S_{i-1})}$;
        $T_{i,j} = max(0, L_{i,j}, U_{i,j}, D_{i,j})$;
// resultant score is the maximum value in T
```

In Sec. I, we introduced two vectoring strategies to reconstruct the data dependencies. Here we describe the major differences between the two vectorizing strategies: (1) *iterate* [6], which **partially** ignores the vertical dependencies in Fig. 1, and processes the vertical cells simultaneously along the column[1] and (2) *scan* [7], which **completely** ignores the vertical dependencies at the beginning. The vertical cells can be processed in a SIMD way, giving us the preliminary results. After that, a parallel max-scan operation is conducted on the preliminary results, and the scan results are applied to correct the results in another round of computation. The fundamental difference in these two strategies is in the correction. While *iterate* may not need any correction or will finish the correction with one or several steps of re-computations, *scan* will always take two rounds of re-computations, i.e., the scan on all vertical cells and then a round of much lighter correction.

Comparing the vector codes in Alg. 2 and Alg. 3[2] to the sequential code in Alg. 1, we see that writing vector codes involves expert knowledge of the algorithms and the platform-specific ISAs, even though the detailed low-level intrinsics are hidden by our formalized codes. As a result, the first question we want to answer is whether we can automatically vectorize these types of applications with multiple combinations of parameters.

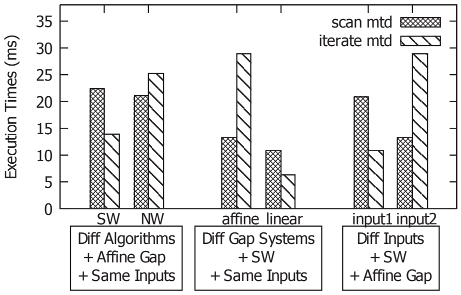


Figure 2: Example of comparing two vectorizing strategies under various conditions on MIC (the cases are from Sec. VI)

On the other hand, the differences in the two strategies each have their own benefits. Fig. 2 shows our other motivation — because the algorithms, configurations, and input sequences at runtime can affect performance and because no one combination can always provide best performance, the second question is whether we can design a mechanism to automatically select the favorable vectorization strategies at runtime.

IV. Generalized Pairwise Alignment Paradigm

Here we present our generalized paradigm for the pairwise sequence alignment algorithms with adjustable gap penalties. Any sequential codes following the paradigm can be processed by our framework to generate real vector codes.

$$T_{i,j} = max \begin{cases} 0 \\ max_{0 \leqslant l < j}(T_{i,l} + \theta_{i,l} + \sum_{k=l+1}^{j} \beta_{i,k}) \\ max_{0 \leqslant l < i}(T_{l,j} + \theta'_{l,j} + \sum_{k=l+1}^{j} \beta_{k,j}) \\ T_{i-1,j-1} + \gamma_{i,j} \end{cases} \tag{2}$$

In the paradigm in Eq.(2), the T is the working-set table and $T_{i,j}$ stores the suboptimal score. 0 is optional and used only in local alignment. $\theta_{i,l}$ ($\theta'_{l,j}$) is the gap penalty of initiating a gap at the position l of Q_m (S_n). $\beta_{i,k}$ ($\beta'_{k,j}$) is the gap penalty of continuing a gap at the position k of Q_m (S_n). $\gamma_{i,j}$ is the substitution score of matching base j of Q_m and base i of S_n. In bioinformatics, the substitution scores are usually from the scoring matrix, such as BLOSUM62. Both $\theta_{i,l}$ ($\theta'_{l,j}$) and $\beta_{i,k}$ ($\beta'_{k,j}$) can be configured to be either constants or variables. By using the dynamic programming, one can use three assistant symbols, i.e., $U_{i,j}$, $L_{i,j}$, $D_{i,j}$, to represent the influence from $T_{i,j}$'s upper, left, and diagonal

[1] This round of computations only ensures a portion of the results as correct, leading to potentially multiple rounds of corrections.

[2] Although we use our formalized codes as the examples, the hand-written vector codes presented in previous research, e.g., [6], are similar to ours.

neighbors. Therefore, the paradigm is equivalent to Eq.(3-6).

$$T_{i,j} = max \begin{cases} 0 \\ U_{i,j} \\ L_{i,j} \\ D_{i,j} \end{cases} \tag{3}$$

$$U_{i,j} = max \begin{cases} U_{i,j-1} + \beta_{i,j} \\ T_{i,j-1} + \theta_{i,j-1} + \beta_{i,j} \end{cases} \tag{4}$$

$$L_{i,j} = max \begin{cases} L_{i-1,j} + \beta'_{i,j} \\ T_{i-1,j} + \theta'_{i-1,j} + \beta'_{i,j} \end{cases} \tag{5}$$

$$D_{i,j} = T_{i-1,j-1} + \gamma_{i,j} \tag{6}$$

Now, we can fit the real algorithms into the paradigm. *Smith-Waterman*: Because it is a local alignment algorithm, we need to keep 0 as the initial. If we simply use the linear gap penalty, the $\theta_{i,l}$ ($\theta'_{l,j}$) is set to 0 and $\beta_{i,k}$ ($\beta'_{k,j}$) is the gap penalty value. If we use affine gap penalty, the $\theta_{i,l}$ ($\theta'_{l,j}$) is the gap open penalty value and $\beta_{i,k}$ ($\beta'_{k,j}$) is the gap extension penalty value. If these parameters are variables, other gap penalty systems can be used. *Needleman-Wunsch*: Because it is a global alignment algorithm, we don't need the 0. The configuration of other parameters is similar with the SW. Actually, ln. 7 to ln. 10 in Alg. 1 follow the paradigm with necessary initialization codes in ln. 1 to ln. 4.

V. AAlign Framework

The AAlign framework adopts the "striped-iterate" and "striped-scan" as the basic vectorization strategies. We make a few modifications to the original methods derived from [6] and [7] to fit our framework. Fig. 3 illustrates the overview of AAlign. The framework can accept any kind of sequential codes following our generalized paradigm in Sec. IV. After analyzing the Abstract Syntax Tree (AST) of the sequential code, AAlign can obtain the required information, such as the type of the given alignment algorithm and the selected gap penalty system. Then, AAlign will input the information to the "vec code constructs" which are formalized according to the aforementioned vectorizing strategies. Finally, the framework can generate real codes by using proper vector modules. These modules include primitive vector operations whose implementation is ISA-specific.

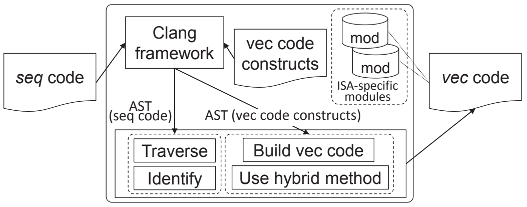


Figure 3: High-level overview of the structure of AAlign framework

A. Vector Code Constructs

In this section, we will first describe the SIMD-friendly data layout used in AAlign. Based on it, we will present two vector code constructs containing the vector modules (Sec. V-C) and the configurable parameters (Sec. V-D).

Striped layout: AAlign always conducts the tabular computation along the query sequence Q_m. After loading the data from the same column in Fig. 1 to the buffer, AAlign transforms the data layout to the striped format, which is SIMD-friendly because the data dependency among adjacent elements are eliminated. Fig. 4 shows the data layouts before and after the striped transformation. In the original buffer, we have 20 elements from the same column of the tabular; and each element depends on its preceding neighbor (the vertical direction in Fig. 1). If we load the elements directly into five vectors, the data dependencies will hinder efficient vector operations. By rearranging the buffer into the striped format, dependent elements are distributed to different vectors, making the interaction happening among vectors rather than within vectors.

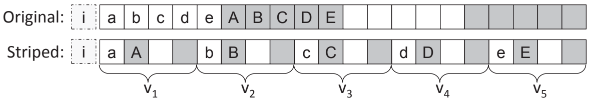


Figure 4: The original and SIMD-friendly striped layouts

Algorithm 2: Vector code constructs for striped-iterate

```
/* m is the aligned length of Q, n is the length of S,
   k is number of vectors in Q, equal to m/vec_len. If
   the linear gap penalty system is taken, the AAlign
   will ignore the asterisked statements                */
vec $vT_{dia}$, $vT_{left}$, $vT_{up}$, vT;
vec $vT_{max}$ = broadcast(INT_MIN);
vec $vGapT_{left}$ = broadcast(GAP_LEFT);
vec $vGapT_{up}$ = broadcast(GAP_UP);
*vec vL, vU;
*vec vGapL = broadcast(GAP_LEFT_EXT);
*vec vGapU = broadcast(GAP_UP_EXT);
*vec vZero = broadcast(0);
for $i \leftarrow 0; i < n; i{+}{+}$ do
    $vT_{dia}$ = $rshift\_x\_fill(arr_{T1} + (k-1) * vec_{len}, 1, INIT\_T)$;
    $vT_{up}$ = $set\_vector(m, INIT\_T, GAP\_UP)$;
    $vT_{up}$ = $add\_vector(vT_{up}, vGapT_{up})$;
    *vU = $set\_vector(m, INIT\_U, GAP\_UP\_EXT)$;
    *vU = $add\_vector(vU, vGapU)$;
    *vU = $max\_vector(vU, vT_{up})$;
    for $j \leftarrow 0; j < k; j{+}{+}$ do
        $vT_{dia}$ = $add\_array(prof + ctoi(S_i) * m + j * vec_{len}, vT_{dia})$;
        $vT_{left}$ = $add\_array(arr_{T1} + j * vec_{len}, vGapT_{left})$;
        *vL = $add\_array(arr_L + j * vec_{len}, vGapL)$;
        *vL = $max\_vector(vL, vT_{left})$;
        *$store\_vector(arr_L + j * vec_{len}, vL)$;
        vT = $max\_vector(vT_{dia}, MAX\_OPRD)$;
        $store\_vector(arr_{T2} + j * vec_{len}, vT)$;
        $vT_{max}$ = $max\_vector(vT_{max}, vT)$;
        $vT_{dia}$ = $load\_vector(arr_{T1} + j * vec_{len})$;
        $vT_{up}$ = vT;
        $vT_{up}$ = $add\_array(vT_{up} + vGapT_{up})$;
        *vU = $add\_vector(vU + vGapU)$;
        *vU = $max\_vector(vT_{up}, vU)$;
    REC_UP = $rshift\_x\_fill(REC\_UP, 1, REC\_FILL)$;
    int $j$ = 0;
    vT = $load\_vector(arr_{T2} + j * vec_{len})$;
    while $influence\_test(REC\_UP, REC\_CRT)$ do
        vT = $max\_vector(vT, REC\_UP)$;
        $store\_vector(arr_{T2} + j * vec_{len}, vT)$;
        $vT_{max}$ = $max\_vector(vT_{max}, vT)$;
        REC_UP = $add\_vector(REC\_UP, REC\_UP\_GAP)$;
        if $++j >= k$ then
            REC_UP = $rshift\_x\_fill(REC\_UP, 1, REC\_FILL)$;
            $j$=0;
        vT = $load\_vector(arr_{T2} + j * vec_{len})$;
    $swap(arr_{T1}, arr_{T2})$;
```

Striped-iterate: This vectorizing strategy is based on [6]. The modified vector code constructs are shown in Alg. 2. We use two m-element buffers $\mathsf{arr}_{\mathsf{T1}}$ and $\mathsf{arr}_{\mathsf{T2}}$ to store the best-matching scores. Additionally, a m-element buffer $\mathsf{arr}_{\mathsf{L}}$ stores the scores denoting best-matching with ending gap in Q. The scores denoting best-matching with ending gap in S are stored in the vector register T_{up} or vU if affine gap penalty system is taken. In this strategy, we first partially ignore the data dependencies within the buffer (along the Q) and use the predefined vectors (ln. 11 and ln. 13) to set lower bounds. In the predefined vectors (T_{up} or vU), only first elements come from the real initialization expressions ($INIT\_T$ and $INIT\_U$), while other elements are derived from them and corresponding gap penalties (GAP_UP and GAP_UP_EXT). As a result, the first round of preliminary computations (ln. 16 to ln. 29) only ensures the first elements in each vector are correct (a-e cells in Fig. 4).

We need to correct the results if the updated predefined vectors affect the results (ln. 33). The re-computations of correction (ln. 34 to ln. 41) will take at most $\mathsf{vec}_{\mathsf{len}}$-1 times to ensure all the other elements in the vectors are correct. After that, we continue the for loop (ln. 9) to process the next character in S, which corresponds to another column in Fig. 1.

Algorithm 3: Vector code constructs for striped-scan

```
   // m is the aligned length of Q, n is the length of S,
      k is number of vectors in Q, equal to m/$vec_{len}$. If
      the linear gap penalty system is taken, the AAlign
      will ignore the asterisked statements
1  vec $vT_{dia}$, $vT_{left}$, $vT_{up}$, vT;
2  vec $vT_{max}$ = broadcast(INT_MIN);
3  vec $vGapT_{left}$ = broadcast(GAP_LEFT);
4  *vec vL;
5  *vec vGapL = broadcast(GAP_LEFT_EXT);
6  *vec vZero = broadcast(0);
7  for $i \leftarrow 0$; $i < n$; $i$++ do
8      $vT_{dia}$ = $rshift\_x\_fill(arr_{T1} + (k-1) * vec_{len}, 1, INIT\_T)$;
9      for $j \leftarrow 0$; $j < k$; $j$++ do
10         $vT_{dia}$ = $add\_array(prof + ctoi(S_i) * m + j * vec_{len}, vT_{dia})$;
11         $vT_{left}$ = $add\_array(arr_{T1} + j * vec_{len}, vGapT_{left})$;
12         *vL = $add\_array(arr_L + j * vec_{len}, vGapL)$;
13         *vL = $max\_vector(vL, vT_{left})$;
14         $*store\_vector(arr_L + j * vec_{len}, vL)$;
15         vT = $max\_vector(vT_{dia}, MAX\_OPRD)$;
16         $store\_vector(arr_{T2} + j * vec_{len}, vT)$;
17         $vT_{dia}$ = $load\_vector(arr_{T1} + j * vec_{len})$;
18     $wgt\_max\_scan(arr_{T2}, arr_{Scan}, m, INIT\_T, GAP\_UP\_EXT, GAP\_UP)$;

19     for $j \leftarrow 0$; $j < k$; $j$++ do
20         $vT_{up}$ = $load\_vector(arr_{Scan} + j * vec_{len})$;
21         vT = $load\_vector(arr_{T2} + j * vec_{len})$;
22         vT = $max\_vector(vT, vT_{up})$;
23         $vT_{max}$ = $max\_vector(vT_{max}, vT)$;
24         $store\_vector(arr_{T2} + j * vec_{len}, vT)$;
25     $swap(arr_{T1}, arr_{T2})$;
```

Striped-scan: The scan strategy in AAlign is based on the GPU method [7]. We modify it by using the striped format on x86-based platforms, shown in Alg. 3. Similar with the striped-iterate, we define three m-element buffers $\mathsf{arr}_{\mathsf{T1}}$, $\mathsf{arr}_{\mathsf{T2}}$, and $\mathsf{arr}_{\mathsf{L}}$. In addition, an extra buffer $\mathsf{arr}_{\mathsf{scan}}$ is used to store the scan results. In this strategy, we first completely ignore the data dependencies within the buffer (along the Q) to do the tentative computation (ln. 9 to ln. 17). Unlike the striped-iterate, we conduct "weighted" scan over the tentative results $\mathsf{arr}_{\mathsf{T2}}$ and store the scan results to $\mathsf{arr}_{\mathsf{scan}}$ (ln. 18). Finally, we use the values in $\mathsf{arr}_{\mathsf{scan}}$ to correct the results (ln. 19 to ln. 24). After that, we continue to process the next character in S (ln. 7).

B. Hybrid Method

As we discussed in Sec. III, no one combination of the algorithms (SW or NW), vectoring strategies (iterate or scan), gap penalty systems (linear or affine) can always provide optimal performance for different pairs of input sequences. Before we provide a better solution, we investigate the reason under what circumstances a specific combination can win. We test various query sequences, whose lengths range from 100 to 36k characters. We fix the algorithm to SW and the gap penalty system to the affine gap, and change the vectoring strategies. We find that the striped-scan strategy performs better when the number of re-computations in striped-iterate is around 1.5 times more on MIC, and 2.5 times on Haswell (For other combinations of algorithms and gap systems, the ratios are similar due to the similar computational pattern and workloads). Generally, if the best-matching score before the re-computations is high, meaning that the two input sequences may be close to each other, the striped-iterate has to carefully and iteratively check each position with more re-computation steps in order to eliminate the false negative; while in striped-scan, no matter what the matching scores are, the fix number of re-computations are needed. Paradoxically, we cannot rely on this observation to determine which strategy should be taken, because unless we finish the alignment algorithm and get the real matching scores, we don't know how similar or dissimilar in the input pair of sequences, or even in a specific rang of pairs.

In the paper, we propose an input-agnostic hybrid method that can automatically select the efficient vectorizing strategy at the runtime. Our hybrid method starts from the striped-iterate strategy, in which we maintain a counter to record the number of re-computations. When the counter exceeds the configured threshold, the method will switch to the striped-scan. For example, based on the experiments for the combination of SW with the affine gap presented in the previous paragraph, we set the threshold to be 2 for MIC and 3 for Haswell CPU. However, switching back from striped-scan to striped-iterate is nontrivial, because we don't know the amount of re-computations for striped-iterate when the algorithm is working in the striped-scan mode. Alternatively, we design a solution to "probe" the re-computation overhead at a configurable interval *stride*. That way, after processing *stride* characters in the subject sequence using the striped-scan, we tentatively switch back to the striped-iterate and rely on the counter to determine the next switch. Once the counter is above the threshold, we switch back again to the striped-scan for another round of processing *stride* characters. Otherwise, our method will stay in the striped-iterate mode and continue checking the counter.

Fig. 5 shows an example of the hybrid method. If we only rely on the striped-iterate method, the re-computations in the middle part of the subject sequence will kill the performance due to the overhead of re-computations. In contrast, if we only use the striped-scan, the benefits of the head and tail parts in the striped-iterate will be wasted. Our hybrid method uses the counter to find the amount of re-computations is above the threshold around processing the 800-th character, and thus switch to striped-scan method. Then, it will probe the counter periodically by going back to the iterate method until the counter drops below the threshold or the end of the sequence S is achieved.

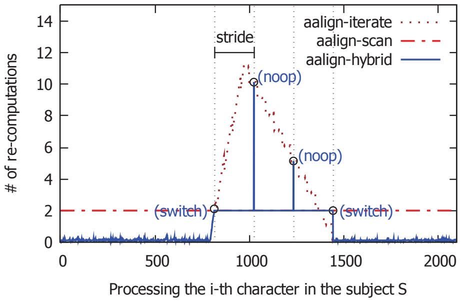


Figure 5: The mechanism of the hybrid method

One may wonder why the hybrid mechanism starts from the striped-iterate, conservatively switches from striped-iterate to striped-scan only when the counter exceeds the threshold, and aggressively switches back by using the proactive probe. The reason is related with the characteristics of sequence search: although the sequence alignment is designed to find similar sequences of databases for the input query, it cannot identify too many similar sequences because statistically most of the sequences of databases are dissimilar with a specific input. Even if a sequence is determined similar to the input, their exactly match regions are few. Considering the much faster convergence speed of striped-iterate for dissimilar pairs, we prefer it, and conservatively switch to striped-scan only when we find current aligned regions are highly matched.

C. Vector Modules

We have already seen the usage of the vector modules in Alg. 2 and Alg. 3. These vector modules are designed to express the required primitive vector operations in our vector code constructs and hide the ISA-specific vector instruction details. Therefore, when the platform changes, AAlign only needs to re-link the vector code to the proper set of vector modules. Tab. I defines the vector primitive modules. The first group of modules are designed to conduct basic vector operations over given arrays or vectors. Specifically, they are wrapper functions of the directly-mapped ISA intrinsics. As a contrast, the second group of modules carry out an application-specific operations, customized to our formalized vector code constructs.

Table I: The vector modules in AAlign

Module Name	Description
Basic Vector Operation API	

load_vector(void **ad*);
store_vector(void **ad*, vec *v*);
	Load/store a vector from/to the memory address *ad*, which can be `char*`, `short*`, or `int*` (the same below)

add_vector(vec *va*, vec *v*);
add_array(void **ad*, vec *v*);
	Add a vector of *va* or from the memory address *ad* by vector *v*,

max_vector(vec *v1*, ...);	Take any count of input vectors, and return the vector with largest integers in each aligned position
App-specific Vector Operation API	
set_vector(int *m*, int *i*, int *g*);	Init a new vector, in which *i* is the default $T_{i,j}$ or $F_{i,j}$ value when j=0, *g* is their corresponding gap $\beta_{i,j}$ or $\theta_{i,j}$

rshift_x_fill(vec *v*, int *n*, ...);
rshift_x_fill(void **ad*, int *n*, ...);
	Right shift the vector of *v* or loaded from *ad* by *n* of positions and fill the gaps with specified values

influence_test(vec *va*, vec *vb*);	Check if vector *va* can affect the values in *vb*
wgt_max_scan(void **in*, void **out*, int *m*, int *i*, int *g*, int *G*);	"weighted" max-scan over the values in *in* of the striped format, store the results to *out*. *i* is the default $T_{i,j}$ value when j=0, g, G are the corresponding $\beta_{i,j}$, $\theta_{i,j}$

set_vector: is to set the lower-bound vector in the striped-iterate strategy. Fig. 6 shows that AAlign will set the first value of the lower-bound vector to be the initial value i Then, the lower-bound values of the rest are set to be $i + l * k * g$, where l is the element's index, k is the total number of vectors, and g is the associate gap penalty. The implementation of the module is to use the proper `_mm256/512_set` instrinsics.

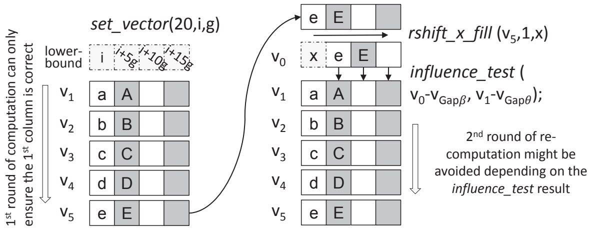

1st round of computation can only ensure the 1st column is correct
set_vector(20,i,g)
lower-bound
i
i+5g
i+10g
i+15g
v1
v2
v3
v4
v5
a A
b B
c C
d D
e E
e E
rshift_x_fill (v5,1,x)
v0
x e E
influence_test (
v0-vGapβ, v1-vGapθ);
2nd round of re-computation might be avoided depending on the influence_test result

Figure 6: Vector modules used in the striped-iterate

rshift_x_fill: is to right shift the vector elements with the value x filled. AAlign uses this module to adjust the data dependencies between vectors. As shown in Fig. 6, the 1st round of computation can ensure the values in the first column (a-e cells) are correct, since they are calculated based on the real initial value i. Therefore, the test of the need for correction is required. Before that, we observe that in the 2nd round, the current "true" value e would affect A according to the original layout in Fig. 4. As a result, we shift the vector v_5 to right by 1 position and fill the gap using a small enough number x to make sure there is no influence caused by it.

The implementation is essentially a combination of data-reordering operations. However, the selection of instructions is quite different because of different ISAs and desired data types. Fig. 7 shows how to achieve the same functionality with different intrinsics. Because the shortest integer data type supported by IMCI is 32-bit, we only show IMCI with 32-bit int, which uses a combination of the cross 128-bit lane *permutevar* and *swizzle* intrinsics. As a contrast, we directly *insert* the value x after the *permutevar* completes on AVX2 with 32-bit int. If we work on the 16-bit values, there is no equivalent *permutevar* intrinsics so that we use *shufflehi/hi*, *permute8x32* and *blend* intrinsics for this functionality, followed by the *insert*.

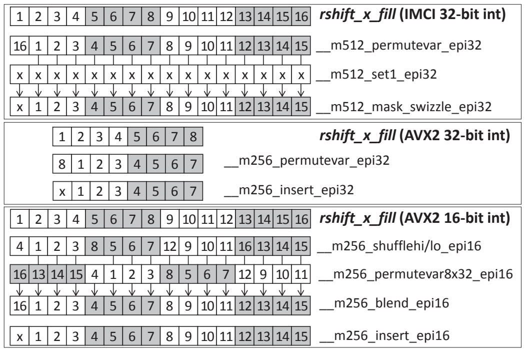


Figure 7: Example of chosen ISA intrinsics for *rshift_x_fill* (only *blend* operations are shown with arrows)

influence_test checks if an extra re-computation of correction is necessary in the striped-iterate method. Specifically, the module is a vector comparison. The comparison results containing 1s mean the 1st operand will affect the 2nd one. In IMCI, the results are stored in a 16-bit mask and then we simply check if this value is larger than 0 or not. However, in AVX2, the "mask" is stored in a 256-bit vector, and there is no single instruction to peek how many set bits inside. Our solution is to split the vector to two 128-bit SSE vectors and use the intrinsic `_mm_test_all_zeros` to detect if there are set bits.

wgt_max_scan implements the "weighted" scan along the buffer holding the tentative results (denoted as $\tilde{T}_{i,j}$ and stored in $\mathsf{arr}_{\mathsf{T1}}$ from ln. 18 of Alg. 3). Mathematically, we perform the calculation of $max_{0\leqslant l<j}(\tilde{T}_{i,l} + \theta_{i,l} + \sum_{k=l+1}^{j} \beta_{i,k})$. For simplicity, let's suppose $\theta_{i,l}$, $\beta_{i,k}$ are two constants θ and β and only use 8 characters as the example for the striped sequence shown in Fig. 4. In Fig. 8, we use three steps to achieve the *wgt_max_scan*. First, we conduct a preliminary round of inter-vector weighted scan on v_1 and v_2 with initial weight $\theta + \beta$ and extensive weight β. The results will be stored in the intermediate vectors u_1 and u_2. Second, an intra-vector and exclusive weighted scan is performed on vector u_2 with the weight of $k * \beta$, where k is the total number of vectors. The results are stored in s. Third, the last round of inter-vector and exclusive weighted broadcast is performed on s, u_1 and u_2 with the weight of β. The final scan results are stored in $\mathsf{arr}_{\mathsf{T1}}$.

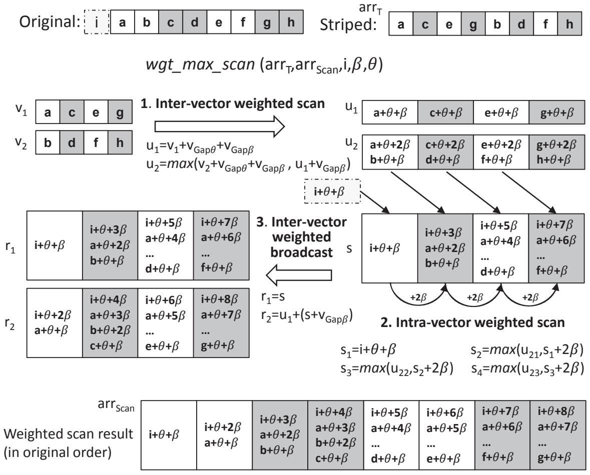

Original:
i a b c d e f g h
arr$_T$
Striped:
a c e g b d f h
wgt_max_scan (arr$_T$,arr$_{Scan}$,i,β,θ)
v$_1$ a c e g
v$_2$ b d f h
1. Inter-vector weighted scan
u$_1$=v$_1$+v$_{Gap\theta}$+v$_{Gap\beta}$
u$_2$=*max*(v$_2$+v$_{Gap\theta}$+v$_{Gap\beta}$, u$_1$+v$_{Gap\beta}$)
u$_1$ a+θ+β c+θ+β e+θ+β g+θ+β
u$_2$ a+θ+2β b+θ+β | c+θ+2β d+θ+β | e+θ+2β f+θ+β | g+θ+2β h+θ+β
i+θ+β
s i+θ+β | i+θ+3β a+θ+2β b+θ+β | i+θ+5β a+θ+4β ... d+θ+β | i+θ+7β a+θ+6β ... f+θ+β
+2β +2β +2β
2. Intra-vector weighted scan
s$_1$=i+θ+β s$_2$=*max*(u$_{21}$,s$_1$+2β)
s$_3$=*max*(u$_{22}$,s$_2$+2β) s$_4$=*max*(u$_{23}$,s$_3$+2β)
3. Inter-vector weighted broadcast
r$_1$=s
r$_2$=u$_1$+(s+v$_{Gap\beta}$)
r$_1$ i+θ+β | i+θ+3β a+θ+2β b+θ+β | i+θ+5β a+θ+4β ... d+θ+β | i+θ+7β a+θ+6β ... f+θ+β
r$_2$ i+θ+2β a+θ+β | i+θ+4β a+θ+3β b+θ+2β c+θ+β | i+θ+6β a+θ+5β ... e+θ+β | i+θ+8β a+θ+7β ... g+θ+β
arr$_{Scan}$
Weighted scan result (in original order)
i+θ+β | i+θ+2β a+θ+β | i+θ+3β a+θ+2β b+θ+β | i+θ+4β a+θ+3β b+θ+2β c+θ+β | i+θ+5β a+θ+4β ... d+θ+β | i+θ+6β a+θ+5β ... e+θ+β | i+θ+7β a+θ+6β ... f+θ+β | i+θ+8β a+θ+7β ... g+θ+β

Figure 8: Orchestration mechanism in the *wgt_max_scan* (Maximum operations are applied on each cell)

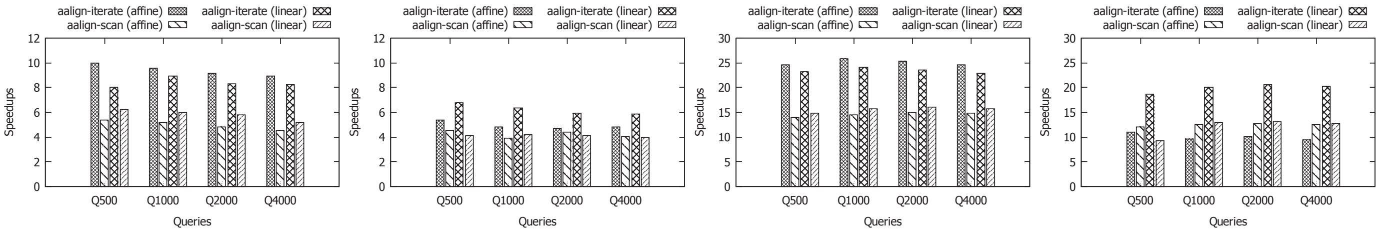
D. Code Translation

The AAlign framework takes the sequential codes following our generalized paradigm as the input. After the analysis of the codes, the framework will decide how to modify the vector code constructs. We make use of Clang driver [10] to create the Abstract Syntax Tree (AST) for both the sequential codes and vector code constructs, shown in Fig. 3. To traverse the trees, we build our Matcher and Visitor classes in Clang's AST Consumer class. Once the information from the AST nodes of interest is identified and retrieved, we use our Rewriter class to modify the AST tree of the vector code constructs with the information and its derivative results. Note, present framework only supports the constant gap penalties (e.g., $\beta_{i,k}$, $\theta_{i,l}$). We will leave it to future work to support variable penalties used in, for example, the dynamic time warping (DTW) algorithm.

Tab. II shows the configurable expressions in Alg. 2 and Alg. 3. The information can be retrieved from the sequential codes in four groups: 1. Identify which type of the pairwise alignment algorithm is used: local or global. This can be done by checking if there is a constant 0 set to T or not. 2. Identify what kind of gap penalty system is used. We can check if θ is set to 0 or not (row 1-4 in Tab.II). 3. Learn how to initialize the boundary values (row 5,6). 4. Derive other information of how to organize the vectors (row 7-11). After the vector code constructs have been rewritten, we use the hybrid method to generate our pairwise sequence alignment kernels.

E. Multi-threaded Version

The AAlign framework can also utilize the thread-level parallelism of the multi- and many-cores to align a given query with all subject sequences in a database. We first assign the generated kernel to each thread, and a thread will get a subject sequence from the database to conduct the alignment until all subject sequences are aligned. After



(a) SW (CPU)

(b) NW (CPU)

(c) SW (MIC)

(d) NW (MIC)

Figure 9: AAlign codes vs. Baseline sequential codes. The baselines are different and they are optimized to follow the similar logic with the corresponding AAlign codes.

Table II: Configurable expressions in vector code contructs

Expression	Description & Format	Example*
GAP_LEFT	Gap penalty from the left T cell (i.e. $\theta'+\beta'$); constants or variables	GAP_{OPEN} (ln.7)
GAP_UP	Gap penalty from the upper T cell (i.e. $\theta+\beta$); constants or variables	GAP_{OPEN} (ln.8)
GAP_LEFT_EXT	Gap penalty from the left L cell (i.e. β'); constants or variables	GAP_{EXT} (ln.7)
GAP_UP_EXT	Gap penalty from the upper U cell (i.e. β); constants or variables	GAP_{EXT} (ln.8)
INIT_T	Upper boundary value of T cell; func(i)	0 (ln.2)
INIT_U	Upper boundary value of U cell; func(i)	0 (ln.2)
MAX_OPRD	Operands required by the max operation; vec variables	vU, vL, vZero
REC_FILL	Value to fill the right shifted gap; constant	GAP_{OPEN} (ln.8)
REC_UP	Operand for checking the re-computation; vec variable	vU
REC_UP_GAP	Gap operand for REC_UP; vec variable	vGapU
REC_CRIT	Criterion for checking re-computation; vec variable	vGapT_{up}-vGapU

*: The examples are fetched or derived from Alg. 1

we sort the database by the subject sequence length, this dynamic binding mechanism is extremely efficient because of the load balance among threads. For the implementation, we don't need to create the profile array of substitution matrix for the query every time (prof in ln. 17 of Alg. 2 or ln. 10 of Alg. 3). Therefore, the only change of the kernel is to extract the part of building profile array and perform it once before launching multiple threads.

VI. Evaluation

In the section, we evaluate the AAlign-generated pairwise sequence alignment codes on Haswell CPU and Knights Corner MIC. For Haswell, we use 2 sockets of E5-2680 v3, which totally contain 24 cores running on 2.5 GHz with 128 GB DDR3 memory. Each core has 32 KB L1, 256 KB L2, and shares 30 MB L3 cache. For MIC, we use the Intel Xeon Phi 5110P coprocessor in the *native* mode. The coprocessor consists of 60 cores running on 1.05 GHz with 8 GB GDDR5 memory, and each core includes 32 KB L1 and 512 KB L2 cache. We use $icpc$ in Intel compiler 15.3 with *-O3* option to compile the codes. To specialize the desired vector ISA, we also include *-xCORE-AVX2* for CPU and *-mmic* for MIC. All the sequences are from NCBI-protein database [11]. The number of characters is integrated into the query name.

Our objectives include the following: (1) compare AAlign-generated codes with the optimized sequential codes; (2) compare our proposed hybrid method with the iterate and scan methods, respectively, and (3) compare multi-threaded versions of AAlign-generated codes with existing state-of-the-art tools.

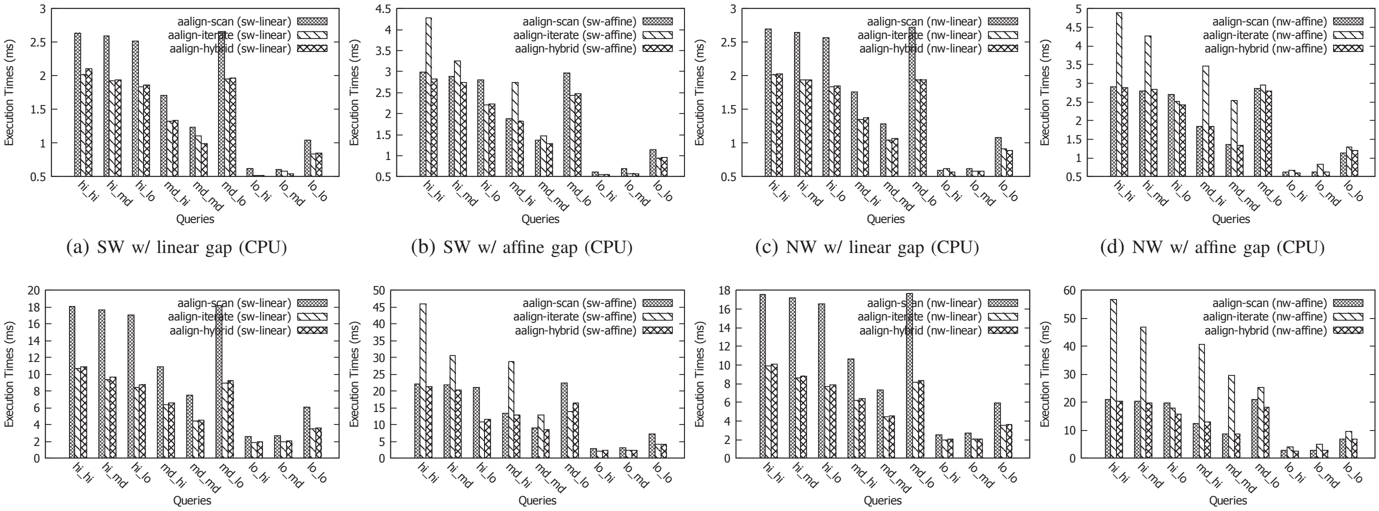
A. Speedups from Our Framework

We first compare the AAlign-generated codes (32-bit int) with the sequential codes (32-bit int) to evaluate the vectorization efficiency. The subject sequence is a Q282. The sequential codes are following the same logic of the vector codes. We also add "#pragma vector always" in the inner-loop of the codes. The speedups, shown in Fig. 9, are the performance benefits brought by the AAlign using striped-iterate and striped-scan respectively. By using the striped-scan, the SW and NW can achieve an average of 4.8 and 13.6-fold speedups over the sequential codes on CPU and MIC respectively. In contrast, the speedups of the striped-iterate SW and NW vary in a wider range of 4.7 to 10-fold on CPU and 9.5 to 25.9-fold on MIC. The superlinear speedups of the striped-iterate are mainly because the striped-iterate avoids a considerable amount of computation along the Q if the $influence\_test$ fails.

We can see that the performance variance of the striped-scan is smaller than the striped-iterate. For example, though the SW approximates the NW in terms of computational workloads, the performance of the striped-iterate SW-affine (Fig. 9c) and NW-affine (Fig. 9d) changes a lot, while the striped-scan keeps relatively consistent. Actually, the performance difference of the two methods depends on the processed numerical values which are affected by the algorithms, gap systems, and input sequences.

B. Performance for Pairwise Alignment

In the preceding section, we observe that the algorithm and gap penalty system will affect the choice of the better vectorizing strategy. This section changes the input sequences. We first borrow the concepts of query coverage (QC) and max identity (MI) [12] from the bioinformatics community to describe the similarity of the input sequences. QC means the percent of query sequence Q overlapping the subject S, while the MI is the percentage of the similarity between Q and S over the length of the overlapped



(a) SW w/ linear gap (CPU)
(b) SW w/ affine gap (CPU)
(c) NW w/ linear gap (CPU)
(d) NW w/ affine gap (CPU)

(e) SW w/ linear gap (MIC)
(f) SW w/ affine gap (MIC)
(g) NW w/ linear gap (MIC)
(h) NW w/ affine gap (MIC)

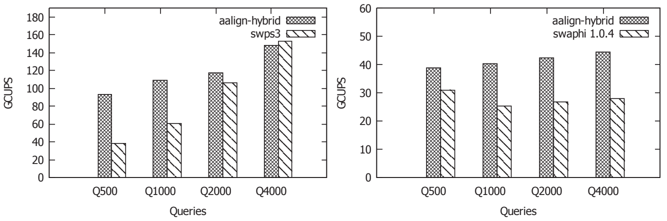
Figure 10: AAlign codes using striped-iterate, striped-scan, and hybrid method. The x-axis represents the similarity of the two sequences using the format of QC_MI in which the query coverage (QC) and max identity (MI) metrics are in three levels: high (>70%), medium (70%-30%), and low(<30%)

area. Additionally, we define three ranges of hi (>70%), md (30%-70%), and lo (<30%). That way, we have nine combinations of QC_MI to represent the similarity and dissimilarity of two input sequences. For example, lo_hi means only a small portion of two sequences overlaps each other, but the overlapped areas are highly similar. In the experiment, we use Q2000 against the "nr" database using NCBI-BLAST [12] and pick out nine typical subjects for the aforementioned criteria.

Fig. 10 shows the performance of AAlign using different vectorizing strategies, including striped-iterate, striped-scan, and hybrid, on CPU and MIC. For the alignment algorithms with linear gap penalty, the striped-iterate method always outperforms the striped-scan, because the effects of the zero θ cause the number of re-computations falling into a very small number. The results also show that with the linear gap penalty, our hybrid method will fall back to the striped-iterate and has very similar performance with it. For the algorithms with affine gap penalty, the striped-scan is better than the striped-iterate when two sequences have high or medium scores of QC and MI, meaning that the input sequences are very similar. For example, for the sequences labeled as *hi_hi*, *hi_md*, *md_hi*, *md_md*, in Fig. 10b, 10d, 10f, and 10h, striped-scan is the better solution, thanks to its fixed rounds of re-computation. In the cases of the NW with the affine gap, the striped-scan can deliver up to 3.5 fold speedup on MIC and up to 1.9 fold speedup on CPU over the striped-iterate. For other inputs (dissimilar input sequences), the striped-iterate is better. Because the hybrid method can automatically switch to the better solution, in most test cases, the hybrid method has better performance than either of the striped-iterate and striped-scan method. In the corner cases, the hybrid method approximates to the better solution instead of the worse one.

C. Performance for Multi-threaded Codes

In the section, we compare AAlign's multi-threaded SW with affine gap penalty system with the tools of SWPS3 and SWAPHI. The database is the "swiss-prot" containing more than 570k sequences [13]. SWPS3 [4] uses a modified version of the striped-iterate method working on CPUs. The buffers of the table T are of char and short data types. SWAPHI [5] supports both inter-sequence and intra-sequence vectorization in the multi-threaded on MIC. In the experiment, we only focus on their intra-sequence method of int data type. Correspondingly, we use our generated kernel of short and int data type on CPU and MIC respectively.



(a) vs. SWPS3 on CPU
(b) vs. SWAPHI on MIC

Figure 11: AAlign Smith-Waterman w/ affine gap vs. existing highly-optimized tools in billion cell updates per second (GCUPS)

Fig. 11 presents the results of AAlign SW algorithms comparing with the two highly-optimized tools. On the CPU, the generated AAlign codes can outperform the SWPS3 for up to 2.5 times, especially for the short query sequences. However, in Fig. 11a, for the long sequences Q4000, SWPS3 is better. This mainly because rather than working entirely on the short data type (16 bits), SWPS3 also uses the char-type (8 bits) buffers. Only when the overflow occurs, the tool will switch to the short. This is especially beneficial for long query sequences by lowering the cache pressure. For the MIC, we can outperform the SWAPHI on an average of 1.6 times, thanks to our hybrid method and the efficient vector modules.

VII. Related Work

To fully utilize the computing power of modern accelerators, it is crucial to utilize the SIMD units within. However, the low programmability are still obstacles facing non-expert programmers. Though some applications can naturally enjoy the benefits brought by the compiler auto-vectorization techniques [14], there are still many applications not belonging to this category. As a result, programmers have to smartly design and hand-code the SIMD codes. [15] propose a fast SIMD sorting algorithm using CPU vector instrinsics. The work of [5], [6], [7] operate on Smith-Waterman by manually writing compiler instrinsics and GPU kernel codes. Heinecke et al. [16] optimize the Linpack Benchmark by using assembly codes on MIC. Unfortunately, explicitly writing vector codes is still not productive and portable. Some compiler-based solutions are proposed to ease the situation. Polyhdral compiler [17] uses a set of loop transformation, optimization and vectorization to generate efficient codes. ISPC [18] provides SIMD-friendly data structures and function APIs. However, these solutions still require the expert knowledge of vectorization and applications.

Other research works focus on the specialized vectorization patterns and code generation. Ren et al. [19] present a set of novel code transformations to facilitate vectorization of recursive programs. PeerWave [20] explores the wavefront parallelism on GPUs including intra-tile parallelism on SIMD units. Ren et al. [21] propose a code generation and optimization engine targeting at using SIMD resources for the irregular data-traversal applications. ASPaS [22] are designed to generate optimized and efficient vector codes for the sorts. Compared to the existing work, the distinctive aspects of our work are to automatically generate the vector codes based on different vectorizing strategies. Our solution is able to switch among these strategies no mater the selected algorithms, configurations, and inputs in the runtime. In addition, our codes are portable among different x86-based systems.

VIII. Conclusion

The AAlign framework can generate the vector codes based on "striped-iterate" and "striped-scan". Moreover, we design an input-agnostic hybrid method, which can take advantage of both the vectorization strategies. The generated codes will be linked to a set of platform-specific vector modules. To do this, the AAlign only needs the input sequential codes following our generalized paradigm. The results show that the vector codes can deliver considerable performance gains over the sequential counterparts by utilizing the data-level parallelism and decreasing the amount of computation. We also demonstrate that our hybrid method is able to automatically switch to the better vectorization strategy at runtime. Finally, compared to the existing highly-optimized multi-threaded tools, the multi-threaded AAlign codes can also achieve competitive performance.

Acknowledgement

This research was supported in part by NSF-BIGDATA program via IIS-1247693.

References

[1] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences," *Journal of molecular biology*, 1981.

[2] S. B. Needleman and C. D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *Journal of molecular biology*, 1970.

[3] P. Rice, I. Longden, A. Bleasby *et al.*, "Emboss: the european molecular biology open software suite."

[4] A. Szalkowski, C. Ledergerber, P. Krhenbhl, and C. Dessimoz, "SWPS3 fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and 86/SSE2," *BMC Res Notes*, 2008.

[5] Y. Liu and B. Schmidt, "SWAPHI: Smith-waterman protein database search on Xeon Phi coprocessors," in *the Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, 2014.

[6] M. Farrar, "Striped Smith-Waterman Speeds Database Searches Six Times over other SIMD Implementations," *Bioinformatics*, 2007.

[7] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the Smith-Waterman Algorithm using Single and Multiple Graphics Processors," *Journal of Computational Physics*, 2010.

[8] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*, 1st ed. Apress, 2013.

[9] Intel. Intel Architecture Instruction Set Extensions Programming Reference. Document ID: 319433-023.

[10] Clang: a C Language Family Frontend for LLVM. http://clang.llvm.org/.

[11] NCBI-protein. http://blast.ncbi.nlm.nih.gov/protein,.

[12] BLAST. http://blast.ncbi.nlm.nih.gov/Blast.cgi.

[13] UniProt: Universal Protein Resource. http://www.uniprot.org/.

[14] K. Hou, H. Wang, and W. chun Feng, "Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study," in *The 43rd IEEE Int'l Conf. on Parallel Processing Workshops (ICCPW)*, 2014.

[15] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture," *Proc. of the VLDB Endowment (PVLDB)*, 2008.

[16] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey, "Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems based on Intel Xeon Phi Coprocessor," in *the IEEE Int'l Symp. on Parallel & Distributed Processing (IPDPS)*, 2013.

[17] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When Polyhedral Transformations Meet SIMD Code Generation," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[18] M. Pharr and W. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *Innovative Parallel Computing (InPar)*, 2012.

[19] B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni, "Efficient Execution of Recursive Programs on Commodity Vector Hardware," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2015.

[20] M. E. Belviranli, P. Deng, L. N. Bhuyan, R. Gupta, and Q. Zhu, "PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization," in *Proc. of the ACM on Int'l Conf. on Supercomputing (ICS)*, 2015.

[21] B. Ren, T. Mytkowicz, and G. Agrawal, "A Portable Optimization Engine for Accelerating Irregular Data-Traversal Applications on SIMD Architectures," *ACM Trans. Archit. Code Optim. (TACO)*, 2014.

[22] K. Hou, H. Wang, and W.-c. Feng, "ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors," in *Proc. of the ACM Int'l Conf. on Supercomputing (ICS)*, 2015.