



Filtr: Compiling Bioinformatics Recurrences

BALA VINAITHIRTHAN, Stanford University, USA
SHIV SUNDRAM, Stanford University, USA
SNEHA GOENKA, Princeton University, USA
FREDRIK KJOLSTAD, Stanford University, USA

Many bioinformatics algorithms, such as sequence alignment and structure prediction, can be expressed as recurrence equations over a dynamic programming matrix. Efficient implementations of these algorithms for large-scale biological data often require changing the order in which matrix cells are calculated and pruning ineffectual regions of the matrix from consideration altogether, but these techniques typically complicate implementation. We introduce Filtr, a domain-specific language (DSL) and compiler framework for bioinformatics recurrences. Filtr keeps the core recurrence rules separate from the pruning and scheduling strategies, where pruning acts as an approximation to limit where in the DP matrix cells are computed, and scheduling determines the iteration order for how cells are explored. Filtr compiles these high-level descriptions into optimized C++ code that matches the performance of hand-tuned implementations while enabling rapid exploration of new heuristics. Filtr is competitive with hand-optimized sequence-alignment libraries, ranging from 0.95× to 30× faster across biological benchmarks.

CCS Concepts: • **Applied computing** → **Bioinformatics**; • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: compilers, domain-specific languages, dynamic programming, bioinformatics, sequence alignment, scheduling, pruning, optimization, locality

ACM Reference Format:
Bala Vinaithirthan, Shiv Sundram, Sneha Goenka, and Fredrik Kjolstad. 2026. Filtr: Compiling Bioinformatics Recurrences. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 372 (October 2026), 30 pages. https://doi.org/10.1145/3839504

1 Introduction

Living organisms are built from sequences of recurring sub-components. DNA consists of repeating nucleotides, genes contain recurring sequences, proteins are chains of repeating amino acids, and larger structures like cells and tissues are composed of recurring molecular building blocks [15, 42, 45]. Hence, modeling bioinformatics problems is often reduced to identifying and comparing recurring sequences and structural patterns [3, 38, 62, 74, 82]. Examples include comparing two DNA sequences to see how similar they are [3, 11], predicting how an RNA folds [49, 50, 57, 89], identifying shared patterns across genes [12, 40, 58, 85], and inferring evolutionary relationships between species [18, 25, 30, 64].

Recurrences decompose sequence modeling problems into overlapping sub-problems. A recurrence equation defines a sequence where each value is calculated in terms of its preceding terms. A classic example from bioinformatics is the Smith Waterman algorithm [68] that determines local alignment by identifying similar regions between a reference and a query sequence. Another example is the Needleman Wunsch algorithm [56] that finds the similarity between sequences of

Authors' Contact Information: Bala Vinaithirthan, Stanford University, Stanford, USA, bala@cs.stanford.edu; Shiv Sundram, Stanford University, Stanford, USA, shiv1@stanford.edu; Sneha Goenka, Princeton University, Princeton, USA, goenka@princeton.edu; Fredrik Kjolstad, Stanford University, Stanford, USA, kjolstad@stanford.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/10-ART372
https://doi.org/10.1145/3839504

comparable length such as complete genes or proteins. Both are ubiquitous in practice and form the crux of critical computational pipelines [3, 7, 38–40]. Recurrences are typically implemented with dynamic programming (DP). In both Smith Waterman and Needleman Wunsch, the core operation is a two-dimensional DP that generates an optimal *alignment* by computing a matrix of possible edits, in which individual DP cells track the minimum cost of replacing, deleting, or inserting characters to morph, or *align*, the query string into the reference string.

Real-world deployment of bioinformatics sequence algorithms relies on both semantics-preserving performance optimization and non-semantics-preserving algorithmic approximation to scale to millions or, in the case of the human genome, billions of base pairs [3, 9, 55, 63, 74]. Moreover, different application pipelines require different accuracy thresholds—for example, finding a mutation requires the exact differences, while finding the regions onto which subsequences map requires only approximate matching. Since the design space of dynamic programs in bioinformatics is large and varied, bioinformaticians have invested substantial effort in creating optimized variants of alignment algorithms. Variants may be designed to algorithmically prune large parts of the computation [9, 13, 17, 20, 29, 31, 32, 47, 48, 76, 87, 88], to make more efficient use of the underlying hardware [2, 4, 5, 26, 43, 44, 53, 59, 69, 72, 80, 84], or a combination of both. The global alignment algorithm, for example, admits several variants, such as banded alignment where only the band around the primary diagonal is computed as this is the most relevant part [9, 76]. Even for this particular *pruned* variant, many important optimization decisions remain. For instance, computing the matrix row by row exposes little parallelism but exhibits good cache locality, whereas computing along antidiagonals (the diagonals running from the bottom-left to the top-right) enables parallelization and vectorization at the cost of poorer cache behavior [22, 83].

Implementations of bioinformatics sequence algorithms that are widely used in practice are hand-written and bundled in large libraries of different variants for different situations [14, 27, 31, 39, 40, 61]. These implementations entangle algorithmic implementation, optimizations, and pruning approaches and must typically be rewritten for new pruning approaches or new platforms. Researchers have also developed programming systems for recurrence problems. The closest to our work are Recuma [71] and Bellman's Gap [65]. Recuma compiles recurrences over dense and sparse arrays but lacks support for algorithmic approximation through pruning optimizations as well as the diagonal iteration pattern that enables vectorized implementations. Bellman's Gap is a logic programming system whose focus is on reducing exponential search to polynomial algorithms. Like Recuma, it does not support diagonal iteration or dynamic pruning, although it has limited support for pruning by placing a static band around the diagonal.

In this paper, we show how to compile a simple but expressive recurrence language to efficient C++ code that implements the machine-level optimizations and approximate pruning optimizations that are used in modern bioinformatics. The optimizations are specified with two distinct optimization languages: a semantics-preserving scheduling language that controls both the order of iteration and the order of result storage, and a semantics-breaking pruning language that controls what parts of the iteration space to prune, presumably because it does not sufficiently contribute to the result. By separating the specification of recurrences from the specification of their optimization, a user can write a recurrence once in a simple syntax and then optimize it for different machines and data sets. Our technical contributions are:

- A recurrence-based intermediate representation that can express a wide range of bioinformatics recurrences along with iteration and pruning optimizations.
- An iteration order scheduling language that supports shearing and value-based iteration (search).
- A pruning language that applies semantics-breaking heuristic optimizations.

- A compiler that applies the iteration order and pruning languages to the Recurrence IR and produces efficient C++ code.

We implemented our ideas in the Filtr compiler for bioinformatics recurrences, which generates native C++ code. Compiled recurrences achieve performance speedups of 0.95× to 30× against recurrence compilers and hand-optimized, often hand-vectorized libraries. We also show that Filtr can explore combinations of optimizations to find the best configuration for a given dataset. Filtr's composable design further enables optimizations unsupported by existing systems: the same pruning program, written once, can be applied across recurrences to produce novel pruned variants that would otherwise require separate hand-written implementations.

2 Motivating Example

Sequence alignment is a core algorithm in bioinformatics that captures how genomic sequences diverge across organisms or across evolutionary time, through three fundamental mutational events: substitution, insertion, and deletion. Regardless of the optimizations applied in a particular use case, the core computation is a series of edit distances expressed as the recurrence:

$$F_{i,j} = \min \begin{cases} F_{i-1,j-1} + [q_i \neq r_j], & \text{match or mismatch (substitution)} \\ F_{i-1,j} + 1, & \text{deletion} \\ F_{i,j-1} + 1, & \text{insertion,} \end{cases}$$

where $[q_i \neq r_j]$ denotes an Iverson bracket whose value is 1 when the query sequence q and reference sequence r mismatch (differ) and 0 when they match.

Figure 1 shows the dynamic programming matrix F that results from computing the edit distance. Each cell (i, j) in the matrix represents the minimum cost of transforming the $1 \ldots i$ prefix of the query string (labeled as q) on the vertical axis into the $1 \ldots j$ prefix of the reference string (labeled as r) on the horizontal axis. As specified by the edit distance recurrence, this cost decomposes into three operations—a match/mismatch (diagonal move), an insertion (horizontal move), or a deletion (vertical move)—where each corresponds to a genetic variation: substituting, inserting, or deleting a single base pair (A, C, T, or G) in the sequence. Thus, each matrix entry or DP cell represents how the recurrence *jointly arrives* at the prefixes $r_{1\ldots j}$ and $q_{1\ldots i}$ from smaller prefixes. The shaded cells and traceback path encode an optimal alignment—a sequence of three mutations (insertions, deletions, and substitutions) that transform the query string into the reference string, with the final edit distance equal to the number of mutations. An insertion corresponds to introducing a gap (denoted "-") in the query and a deletion corresponds to introducing a gap in the reference. While this recurrence equation fully specifies the computation, production aligners rarely evaluate the full $O(N^2)$ DP as this is prohibitive for genomics pipelines that run such a DP over millions of separate sequence pairs [34, 39, 78]. For example, aligning one million 1,000-base pair sequences requires 10^{12} DP cell computations.

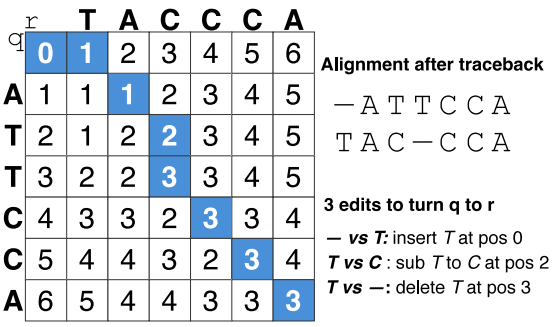
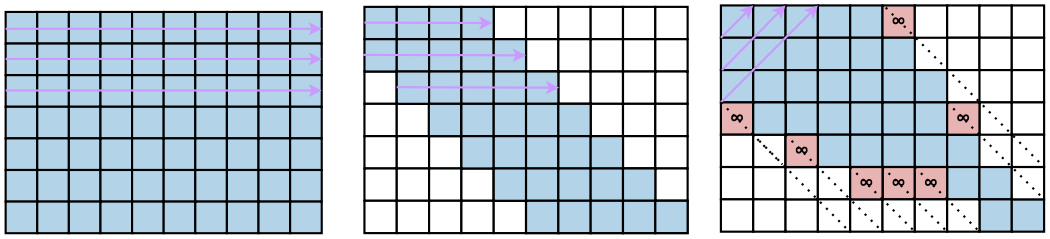


Fig. 1. Edit distance example.

Therefore, real-world implementations optimize by controlling the order of computation and by pruning out work that is unlikely to contribute to the result. Figure 2 shows three variants of the edit distance recurrence. These approaches share the same recurrence equation but have very



(a) Standard alignment. (b) Banded alignment. (c) X-drop alignment.

Fig. 2. Three edit distance alignment variants that differ in where cells are computed and what order. Blue cells indicate the active region. (a) Row-major standard alignment. (b) Banded alignment restricts computation to a static diagonal band. (c) X-drop proceeds in antidiagonal order, marks cells as ∞ if they are unlikely, and prunes along the diagonal boundary established by ∞, producing data-dependent, dynamic pruning.

different implementations due to different optimization strategies. The first variant (Figure 2a) is a standard vanilla implementation that executes the whole recurrence in row-major order.

The second variant (Figure 2b) is the banded alignment algorithm used in BWA-MEM [40] and other production aligners [14, 31, 39, 41]. Related biological sequences tend to align near the main diagonal of the dynamic programming matrix because large, sustained insertions or deletions are rare [9, 52]. Intuitively, aligning identical sequences would result in a path that lies exactly along the main diagonal of the matrix because there are no insertions and deletions. In this case, only diagonal cells would need to be computed. Assuming sequences are similar but not identical, only cells near the main diagonal must be explored. Banded alignment exploits this observation by restricting evaluation to a fixed-width diagonal band, as shown by the DP cells colored in Figure 2b.

The third variant (Figure 2c) is the X-drop alignment algorithm used in BLAST [3], minimap2 [39], and others [31, 61, 72, 84]. When alignments contain long stretches of insertions or deletions, such as the alignments found in evolutionary analysis between distantly related species, a fixed band may be too narrow to capture the optimal alignment or too wide to be fast. At the cost of more complicated control flow and data-dependent behavior, X-drop improves on fixed banding by making the pruning data dependent. Rather than assuming a fixed diagonal bound, X-drop allows the alignment to dynamically determine which regions are promising based on the observed scores and which can be ignored. X-drop processes the alignment one antidiagonal at a time. As the recurrence advances, the algorithm tracks the best score seen on the previous antidiagonal. Any diagonal whose score falls more than a threshold X below this score is marked inactive (shown as a red ∞ in Figure 2c) and is not extended further along the diagonal (dotted line).

Methods such as above have been developed in bioinformatics to avoid evaluating the complete matrices, especially as the number of alignments [39] or length of the sequences increase significantly [31]. These make the computational time tractable while giving up minimal accuracy. As a result, most of the implementation effort goes into reducing the number of matrix cells evaluated and the order of evaluation, not writing the recurrence itself. This creates two problems. First, trying new pruning heuristics typically means reimplementing the whole algorithm, which discourages experimentation. Second, performance optimizations must be repeated for each pruning variant.

3 The Recurrence Language

Bioinformaticians reason about algorithms in terms of their recurrence equations. These equations are the shared language of the field, appearing in papers and lectures as the canonical description of how an algorithm models biological reality [18, 30, 56, 68].

$$
\begin{aligned}
\text{Prog} &::= \text{Rec}^{+} \\
\text{Rec} &::= \text{TA} = \text{Expr} \\
\text{TA} &::= t[\text{Idx}(,\text{Idx})^{*}] \\
\text{Idx} &::= v + n \quad (n \in \mathbb{Z}) \\
\text{Expr} &::= \text{TA} \mid \mathit{const} \\
&\quad \mid \text{Expr Op Expr} \\
&\quad \mid \text{ReduceOp}_{v,\,\text{Cond}^{*}}\ \text{Expr} \\
&\quad \mid f(\text{Expr}(,\text{Expr})^{*}) \\
\text{Op} &::= + \mid - \mid * \mid / \mid \max \mid \min \\
\text{ReduceOp} &::= \Sigma \mid \Pi \mid \max \mid \min \mid \arg\max \mid \arg\min \\
\text{Cond} &::= \text{Expr CmpOp Expr} \\
\text{CmpOp} &::= < \mid \leq \mid > \mid \geq \mid = \mid \neq
\end{aligned}
$$

Fig. 3. Grammar of recurrence language.

Table 1. Example recurrences for different biology problems.

Topic	Example Recurrence
Sequence Alignment	$M_{i,j} = \max(M_{i-1,j-1},\ I_{i-1,j-1},\ D_{i-1,j-1}) + \mathit{in1}_{i,j}$ $I_{i,j} = \max(M_{i-1,j} - c_1,\ I_{i-1,j} - c_2)$ $D_{i,j} = \max(M_{i,j-1} - c_1,\ D_{i,j-1} - c_2)$
Sequence Chaining	$C_j = \mathit{in1}_j + \max_{i<j} C_i$
RNA Folding	$F_{i,j} = \max(F_{i+1,j},\ F_{i,j-1},\ F_{i+1,j-1} + \mathit{in1}_{i,j},$ $\max_{i \le k < j}(F_{i,k} + F_{k+1,j}))$
Sequence Prediction	$M_{i,j} = \mathit{in1}_{i,j} \cdot \max(M_{i-1,j-1} \cdot c_1,\ I_{i-1,j-1} \cdot c_2,\ D_{i-1,j-1} \cdot c_3)$ $I_{i,j} = \mathit{in2}_i \cdot \max(M_{i-1,j} \cdot c_4,\ I_{i-1,j} \cdot c_5,\ D_{i-1,j} \cdot c_6)$ $D_{i,j} = \mathit{in3}_j \cdot \max(M_{i,j-1} \cdot c_7,\ I_{i,j-1} \cdot c_8,\ D_{i,j-1} \cdot c_9)$

Filtr's recurrence language serves as the primary user-facing input to Filtr, capturing the mathematical structure of a bioinformatics model exactly as described in the literature, without implementation detail about how or where it is evaluated. Filtr's recurrence language is based on the Recuma language [71], and follows the grammar described in Figure 3. The language is made up of one or more recurrence equations, each of which defines the value of a cell in a multidimensional array as a function of previously computed values, input data, and scalar constants. The function can be composed of arithmetic operators, aggregation operators such as max, reductions over index ranges, and side-effect-free user-defined functions.

A set of recurrence equations is valid in Filtr if every dependency points to a cell computed earlier in the iteration order. For example, computing $S_i = P_i$ where $P_i = S_{i-1} + 1$ is admitted because the dependency relies on a past iteration step, as index variables like i proceed from lower to higher values. The equation $S_i = P_i$ where $P_i = S_i$ is rejected due to an immediate cyclic dependency. More generally, Filtr follows the Recuma model of correctness, in which a recurrence's dependencies are statically analyzed within the context of the loop nest that encapsulates the recurrence expression; each dependency must be a value produced earlier in the same loop iteration, or a value produced in a previous loop iteration. By extension, Recuma can handle mutually recursive recurrences, as long as they follow this dependency model. For example, while $S_i = P_{i-1}$ and $P_i = S_{i-1}$ are mutually recursive, they are permissible because P_i and S_i's dependencies are from the previous iteration $i - 1$: P_{i-1} and S_{i-1}. Recuma's dependency analysis is described in detail in Sections 5 and 6 of the original Recuma paper [71].

Table 1 shows representative recurrences from four major classes of bioinformatics problems. Though they model different biological processes, they share several structural properties. First, dependencies follow regular patterns: cells reference neighbors along rows, columns, diagonals, and antidiagonals. Second, many recurrences are mutually defined: affine gap alignment (shown as Sequence Alignment example recurrence in Table 1) and sequence prediction each couple three arrays (M, I, D) that reference one another at each step [19, 28]. Third, some recurrences involve reductions over variable-length ranges—RNA folding maximizes over k, and sequence chaining reduces over all predecessors. In all cases, a global optimization problem decomposes into overlapping subproblems evaluated over a rectangular or triangular index space.

4 System Overview

The Filtr system compiles simple recurrence equations to efficient C++ code. Figure 4 gives an overview of the compilation process. Recurrence equations expressed in the language of Section 3

are lowered to a recurrence intermediate representation (Recurrence IR), which is a more sophisticated recurrence language that can express the optimized iteration order and iteration pruning optimizations needed for state-of-the-art bioinformatics kernels. Two optimization languages (shown in green) are used to specify rewrites on the Recurrence IR. While the Recurrence Language (shown in blue) specifies *what* is computed, the Iteration Ordering Language specifies *how* to order the computation, and the Pruning Language specifies *where* computation should be skipped.

This separation of concern lets a user write down a bioinformatics recurrence using simple intuitive Cartesian coordinates and bounds, and then vary iteration order optimizations and pruning strategies depending on the characteristics of the target machine and sequence data. To illustrate, the base edit distance recurrence, using Cartesian coordinates, is $F_{i,j} = \min(F_{i-1,j-1} + [q_i \neq r_j],\ F_{i-1,j} + 1,\ F_{i,j-1} + 1)$ where $0 \leq i \leq M$ and $0 \leq j \leq N$. The user varies iteration order by providing a list of coordinate indices (e.g., *loopOrd*(i, j)). Because the Cartesian indices i and j map directly to traditional matrix rows and columns, evaluating this form in index order results in a standard row-by-row sweep of the matrix. The equivalent antidiagonal form of this recurrence is $F_{k,m} = \min(F_{k-2,m-1} + [q_{k-m} \neq r_m],\ F_{k-1,m} + 1,\ F_{k-1,m-1} + 1)$ where $0 \leq k = i + j \leq M + N$ and $\max(0, k - M) \leq m = j \leq \min(N, k)$. Following index order, the antidiagonal form sweeps one antidiagonal at a time. Antidiagonal traversal is vectorizable and thus more performant, but more complex for users to express and understand. In Filtr, the user can provide the simpler version of edit distance in the Recurrence Language, and transform it into an antidiagonal form in the Recurrence IR (Section 5) with the command $shear(k = antidiag(i, j), m = j)$ from the Iteration Ordering Language (Figure 4).

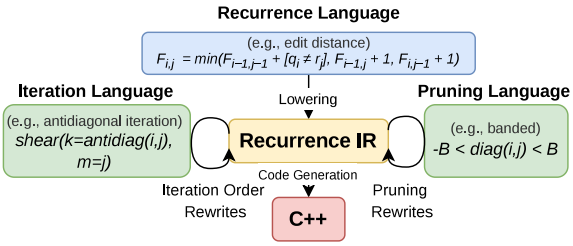


Fig. 4. The recurrence language is lowered to the Recurrence IR, which is then optimized by user-specified iteration and pruning rewrites before C++ code generation. We include an antidiagonal-iterated banded edit distance example.

The *Iteration Ordering Language* (Section 6) lets the user control how a recurrence is executed by specifying the order in which its matrix cells are traversed. It is a semantics-preserving language, which means it guarantees the same final values of the recurrence and the traceback path. As an optimization, the chosen traversal order also automatically determines the storage layout: cells are stored so that consecutively visited cells are contiguous in memory. Storage optimizations arise naturally from traversal strategy and in practice, bioinformatics workloads do not require storage and iteration to be specified independently as is common in, e.g., tensor algebra [35].

The *Pruning Language* (Section 7) is a semantics-breaking approximation language that lets the user remove points in the recurrence's iteration domain from the computation. Users express constraints that define a region of the iteration domain, and only result cells within this region are computed. These constraints may be simple inequalities over the iteration domain coordinates (static pruning) or they may take the form of recurrences over the iteration domain whose values evolve during execution (dynamic pruning).

Both the iteration and pruning languages are rewrites on the RIR. Iteration ordering rewrites run before pruning, because pruning can be written in terms of loop variables that exist only after an iteration order is chosen. For example, a recurrence may be expressed in terms of Cartesian coordinates i and j, but the chosen loop order may involve the new variable $k = antidiag(i, j)$, which can also be used in the pruning language. Each time a rewrite transforms the RIR, the compiler re-checks that the recurrence's dependencies are satisfied in the new coordinate space

using Recuma's dependency checking approach. The validity conditions of each Filtr rewrite are described in Sections 6 and 7.

Filtr has several limitations in scope, optimization, and hardware support. In scope, Filtr compiles individual recurrences rather than a full pipeline. For example, Filtr targets the core dynamic programming kernel of a modern alignment pipeline. Filtr does not address the surrounding pipeline stages, such as the initial seeding stage, which are important but comprise a much smaller percentage of runtime. Filtr also computes over dense matrices with regular access patterns, and cannot express recurrences over sparse matrices. In optimization and hardware, Filtr runs on a single CPU core and leaves vectorization to the C++ compiler. It does not produce code that runs across multiple cores, GPUs, or FPGAs, and it does not support cache blocking. Finally, Filtr stores the full value in every cell and does not use the memory-saving tricks of packing cells into one machine word or storing only the difference between cells. Targeting more architectural backends and supporting more optimizations is left as future work.

5 Recurrence IR

The Recurrence IR (RIR) is a language expressive enough to fully describe an optimized bioinformatics algorithm, including its recurrence logic, its traversal order, and pruning of the iteration domain. The Recurrence Language from Section 3 was deliberately designed to describe only the mathematical model, and to say nothing about how or where the recurrences are evaluated. This simplicity allows the Recurrence Language to express a large number of algorithms from the bioinformatics literature in a simple and intuitive way, but prohibits it from representing the banding, dynamic pruning, and non-standard iteration orders that optimized kernels require.

The RIR extends the Recurrence Language with the grammar in Figure 5 that introduces features to allow for exploration of iteration and pruning strategies—explicit domains and conditional definitions. The RIR allows domain bounds on iteration variables to be expressions over outer index variables and to be defined by their own recurrences. These allow the iteration space to vary with the outer loop index and to contract at runtime based on previously computed values. The RIR also supports conditionally defined recurrence equations and set-builder reductions. A recurrence equation may produce different values depending on a condition, and a reduction may aggregate over indices satisfying a condition. Figure 6 illustrates these additional features through three progressively more complex formulations of edit distance.

```
RIR    ::= TA : Region+
Region ::= Stmt+ domain : Cond+
Stmt   ::= Rec
         | TA = Expr if Cond else Expr
Expr   ::= ...   (as in Figure 3)
         | ReduceOp ( Expr if Cond+ )
```

Fig. 5. Grammar of Recurrence IR, extending the Recurrence Language (Figure 3). Each region carries its own domain conditions, enabling domain splitting. Domain bounds may be expressions over outer index variables and recursive.

An algorithm in the RIR consists of one or more recurrence equations that jointly compute one or more tensors. Each recurrence equation is paired with a domain that specifies the values over which its index variables range. When a recurrence is first lowered from the Recurrence Language, the compiler infers a simple domain from the index variables; the scheduling and pruning languages then embellish both the equations and the domain.

5.1 Recurrence Domains

Every recurrence in the RIR is paired with a domain that specifies the bounds over which its index variables range. Domains in the RIR may be static (e.g., $-3 < i < 3$), or dynamic (e.g., $\text{col_lo}_i < j < \text{col_hi}_i$), where col_lo_i and col_hi_i are dynamically calculated (via separate recurrences) on each new row i.

In the simplest case—a rectangular domain—the bounds are constants. The original edit distance (Figure 6, left) is defined over $0 \le i \le M,\ 0 \le j \le N$. When a recurrence is first lowered from the Recurrence Language, the compiler infers constant bounds on each index variable, producing a simple domain. This works well for straightforward Cartesian formulations.

Many optimized algorithms require more complex iteration spaces, beyond the simple bounds that can be expressed with constants. The RIR thus allows bounds on inner index variables to be symbolic expressions over outer index variables. For instance, antidiagonal-major traversal, where cells are grouped by the sum $k = i + j$, produces a domain in which the number of cells per antidiagonal first grows (until hitting the middle antidiagonal), and then shrinks. The correct bounds, shown in Figure 6 (center), are $\max(0, k-M) \le m \le \min(N, k)$, where m is the position along antidiagonal k. These derived bounds are the result of a Cartesian to diagonal coordinate transform that our compiler handles automatically (Section 6).

Some algorithms require the domain itself to change during execution. X-drop and other dynamic pruning strategies from literature contract the active region based on the actual scores that have been so far calculated. To support this, the RIR allows recurrences to determine the domain, not just the output data. More specifically, *data recurrences* (the recurrences seen up to this point) like edit distance are merely evaluated over cells within a given domain. A *domain recurrence*, on the other hand, is a recurrence whose computed values define the *domain*—the bounds over which data recurrences are evaluated. This makes the iteration space itself programmable—what cells the algorithm computes next can depend on what it has computed so far. The X-drop formulation (Figure 6, right) illustrates this: diag_lo_k and diag_hi_k are domain recurrences that narrow the range of m at each antidiagonal based on which cells were dynamically pruned on previous antidiagonals.

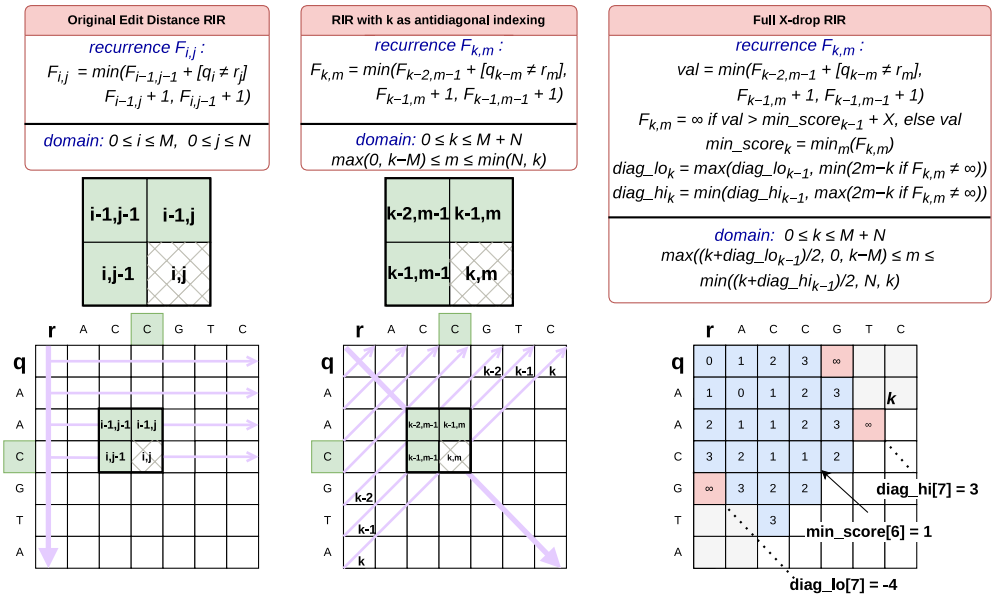


Fig. 6. Progressive refinement of the edit distance Recurrence IR. (Left) The original Cartesian formulation indexes by (i, j) with row-major traversal. (Center) After changing to antidiagonal coordinates (k, m), the same values are computed along antidiagonals. Dependencies are shown in green. (Right) The full X-drop formulation adds conditional and domain recurrences; the active region narrows as cells exceeding the score threshold are set to ∞ and the bounds diag_lo, diag_hi contract accordingly.

The lower bound on m incorporates diag_lo_{k-1}, and the upper bound incorporates diag_hi_{k-1}, so the active region shrinks as computation proceeds.

Domain recurrences and data recurrences may be mutually recursive, meaning their calculations are interdependent. In X-drop, the data recurrence for $F_{k,m}$ determines which cells are given tangible scores or set to ∞; the domain recurrences for diag_lo and diag_hi analyze the spatial distribution of these ∞ values on the current antidiagonal and then update the domain bounds, which define which cells of F will be evaluated on the next antidiagonal. This mutual recursion—in which data values influence future domain bounds, and domain bounds determine which cells are computed—is exactly what enables data-dependent pruning at runtime. The Pruning Language (Section 7) provides the user-facing constructs for specifying such strategies to be applied to the RIR.

5.2 Recurrence Equations

In the simplest case, a recurrence equation in the RIR directly computes an array element from previously computed values, exactly as in the Recurrence Language. The original edit distance in Figure 6 (left) is an example. The RIR extends the Recurrence Language with conditional definitions and set-builder reductions.

A recurrence equation may be *conditionally defined*: a single array element is given by one of several expressions, each guarded by a Boolean condition built from comparisons over index variables, constants, input data, or previously computed values. Simple index-only conditions (such as $i = 0$) are useful for representing base cases. More powerfully, conditions that depend on computed values enable pruning within the recurrence itself.

The X-drop formulation in Figure 6 (right) demonstrates this: $F_{k,m} = \infty$ if val $>$ $\text{min_score}_{k-1} + X$, else val. If the candidate score exceeds the best score seen on the previous antidiagonal by more than the X-drop threshold, the cell is pruned by setting it to ∞. In Figure 6, the cells marked ∞ at antidiagonal k=7 are the result of this pruning. Otherwise, the cell keeps its computed value.

The RIR also adds a *set-builder reduction*, which reduces an expression over all indices satisfying a condition. For example, $\text{diag_lo}_k = \max\big(\text{diag_lo}_{k-1},\ \min(\ 2m - k \text{ if } F_{k,m} \neq \infty\ )\big)$. This finds the smallest diagonal index among non-pruned cells on antidiagonal k. The analogous formula for diag_hi_k uses max in place of min to find the largest active diagonal index. These values tighten—via the outer max and min with the previous antidiagonal's bounds—to narrow the active region of the matrix. Together with the data recurrence tracking the min_score, these set-builder reductions complete the data-domain loop that allows for dynamic pruning.

6 Iteration Ordering Language

A recurrence's iteration space is the geometric shape of its domain. Its iteration order is the order in which points in that space are visited. Filtr provides three index-space transformations—*loop order*ing, *shear*ing, and *search*ing—whose grammar is given in Figure 7. These transformations rewrite the recurrence's iteration order and, as an automatic optimization, its storage layout and iteration space. Choosing a traversal order and a storage layout are tightly intertwined decisions because the order in which cells are visited informs which values should be kept in memory and cache. Performance of a recurrence program depends on the order in which cells are evaluated: different traversal orders can expose parallelism, improve memory locality, or eliminate impossible states entirely.

```
Prog ::= Trans*
Trans ::= LoopOrd | Shr | Search
LoopOrd ::= loopOrd(Vars)
Shr ::= shear(CoordDef(, CoordDef)*)
CoordDef ::= v = CoordExpr
Search ::= search(score, CoordDef(, CoordDef)*)
CoordExpr ::= antidiag(Vars) | diag(Vars) | v
Vars ::= v(, v)*
```

Fig. 7. Grammar of Iteration Ordering Language. loopOrd, antidiag, and diag are abbreviations.

Ordering commands are named rewrites on the RIR and leave the recurrence's traceback and final value unchanged, i.e., are correctness preserving. Each transformation takes an RIR as input and produces a rewritten RIR in a new coordinate system. *loopOrd*, the simplest case, swaps the nesting of index variables. *Shearing* introduces new coordinates as linear combinations of the originals, reshaping the iteration space so that independent computations become contiguous. This transformation both changes traversal order and reorders data layout to be antidiagonal major, as opposed to row major or column major. *Search* converts the indices of the original matrix so that position becomes the stored quantity, and score or cost (i.e., edit distance) becomes an index, intuitively converting a DP matrix traversal into a search that visits only possible states. This transformation changes traversal and reorders data layout to be score-major, meaning the traversal follows higher scores.

The *loopOrd* rewrite is allowed when the resulting loop order visits every dependency of a cell before the cell itself. Likewise, *shearing* is allowed when the combined indices, such as the combined antidiagonal index $i + j$, do not break the recurrence's dependency structure. *Search* is possible when the recurrence computes a score using min or max, and the score changes monotonically as it moves from one cell to another, either remaining the same or changing by a fixed amount.

A rewrite whose preconditions are satisfied and that leaves dependencies satisfied produces a well-formed RIR. A sequence of rewrites is therefore valid when each step's precondition holds locally. In the subsequent sections, we first describe the user-facing specification of each transformation, the compiler rewrites that implement it, and the conditions under which the compiler admits it.

6.1 Loop Order

A recurrence permits multiple valid evaluation orders. In edit distance, row-major traversal, iterating over i in the outer loop and j in the inner loop, visits cells row by row, producing good cache locality for row-major storage. This and other valid orderings compute the same values; however, the order in which the DP matrix is filled determines memory access patterns, parallelism, and compatibility with pruning strategies. A loopOrd is valid if the new order maintains the invariant that dependent cells get computed before the current cell that requires them.

Language Feature. The simplest transformation reorders loop nesting. A recurrence over indices (i, j) admits multiple valid evaluation orders. The user specifies the desired nesting with the *loopOrd* construct: *loopOrd(i, j)* produces row-major traversal; *loopOrd(j, i)* produces column-major.

RIR Rewrite. Given a requested loop order, the compiler rewrites the domain by solving the bound constraints for each variable in nesting order: the outermost variable's bounds are solved as constants or expressions independent of inner variables, and each subsequent variable's bounds are solved in terms of the enclosing variables. For simple domains, this amounts to swapping the order of the bound expressions. For more complex domains produced by subsequent shearing transformations (Section 6.2), the compiler re-solves the symbolic inequalities in the new variable order. After rewriting the domain, the compiler verifies that the loop order respects the recurrence's data dependencies. If a user specifies a loop ordering that violates a dependency, the compiler rejects the program with an error.

6.2 Shearing

In the edit distance recurrence, neither rows nor columns offer straightforward parallelism. Each cell (i, j) in the edit distance recurrence depends on $(i-1, j)$, $(i, j-1)$, and $(i-1, j-1)$, neighbors in both the row and column directions. Iterating along a row, every cell depends on the cell immediately to its left; iterating down a column, every cell depends on the cell immediately above it. In either

case, cells within the same loop iteration are linked by dependencies, preventing straightforward parallel evaluation.

However, all three dependencies share a property: they have strictly smaller values of $i + j$. This means that every cell on a given antidiagonal—the set of cells where $i + j$ equals some constant k—depends only on cells from earlier antidiagonals. All cells on the same antidiagonal are mutually independent and can be computed in parallel (Figure 8).

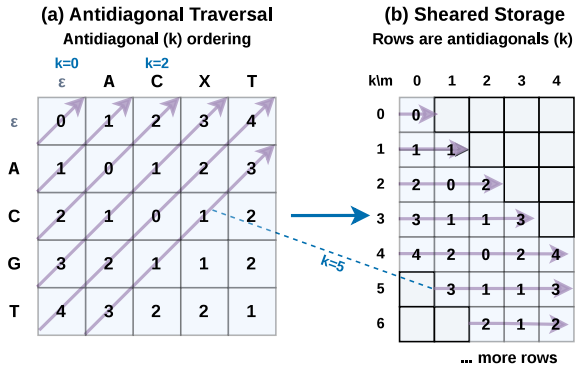


Fig. 8. (a, left) Cells on the same antidiagonal (constant i+j) are independent but lie in different rows. (b, right) After shearing, each antidiagonal maps to a contiguous row in the storage layout. Shearing couples traversal and storage (antidiagonal becomes sequential in memory).

Antidiagonal traversal solves the parallelism problem but introduces a memory locality problem. If the DP matrix is stored in row-major order, consecutive cells on an antidiagonal lie in different rows and typically in different cache lines. Each step along the antidiagonal jumps to a new row, producing strided memory accesses. Filtr controls the storage layout of the temporary dynamic programming matrices and the shearing transformation reorganizes the layout so that the storage order coincides with the iteration order, as this ensures good spatial locality. That is, it both reshapes the iteration space so that antidiagonals are traversed in sequence and also reshapes storage so that each antidiagonal occupies a contiguous block in memory. After shearing, the cells of each antidiagonal are contiguous in memory, so parallel traversal benefits from good locality. Figure 8 illustrates this: panel (a) shows the strided access pattern of antidiagonal traversal over the original matrix, while panel (b) shows the sheared storage layout where each antidiagonal k maps to a contiguous row.

loopOrd (Section 6.1) reorders the loop nest and leaves the storage layout unchanged. *shear* introduces new coordinates, setting both the storage layout and a default traversal order. When the two are combined, *shear* sets the storage layout and *loopOrd* sets the traversal order, replacing the default that *shear* would use.

Language Feature. The user requests a shearing transformation using the *shear* command, where the new coordinates are defined as linear combinations of the original coordinates. For antidiagonal storage in edit distance, the user writes $shear(k = antidiag(i, j),\ m = j)$ where the first coordinate k is the antidiagonal index $i + j$ and the second coordinate m is the position within that antidiagonal.

Filtr provides built-in names for coordinate functions common in bioinformatics: *antidiag(i,j)* groups cells by antidiagonal index, *diag(i,j)* groups cells by diagonal index. Users may also define custom linear combinations, though antidiagonals and diagonals cover the common cases in two-dimensional sequence alignment, sequence prediction, and RNA folding.

RIR Rewrite. The shearing transformation rewrites the RIR from the original coordinate system into the sheared coordinates $(i, j) \mapsto (k, m)$. For edit distance, the RIR before and after the transformation is:

RIR Before:

$$F_{i,j} = \min\big(F_{i-1,j-1} + [\mathrm{q}_i \neq \mathrm{r}_j],\ F_{i-1,j} + 1,\ F_{i,j-1} + 1\big)$$

$$0 \le i \le M, \qquad 0 \le j \le N$$

RIR After:

$$F_{k,m} = \min\big(F_{k-2,m-1} + [\mathrm{q}_{k-m} \neq \mathrm{r}_m],\ F_{k-1,m} + 1,\ F_{k-1,m-1} + 1\big)$$
$$0 \le k \le M + N, \qquad \max(0, k - M) \le m \le \min(k, N)$$

The values computed are identical, but the recurrence expression has completely changed its indexing. Every occurrence of i has been replaced by $k - m$ and every occurrence of j by m. The dependency geometry has changed. In the original RIR, dependencies point to $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$, a mix of row and column neighbors. In the rewritten RIR, all dependencies point to antidiagonals $k - 1$ or $k - 2$. DP cells within the same antidiagonal k are independent. The domain has also changed shape: the original rectangle has become a parallelogram, with inner bounds on m that depend on the outer variable k.

Two features of the RIR enable this transformation. First, the RIR represents domains as symbolic bounds that may depend on outer loop variables, so the non-rectangular iteration space that results from shearing—where the range of m changes at each antidiagonal k—can be expressed. Second, shearing merely modifies the index variables inside array accesses and leaves the surrounding mathematical operations untouched. Therefore, the rewrite applies uniformly across recurrences.

At a high level, the rewrite introduces a new index variable $k = \mathit{antidiag}(i, j)$ representing the antidiagonal, substitutes the original indices with expressions in the new coordinates, and adjusts the loop bounds to reflect the reshaped iteration space. Because shearing is an invertible coordinate change, the compiler can compute an inverse mapping from new coordinates to old. For $\mathit{shear}(k = \mathit{antidiag}(i, j), m = j)$, the inverse is $i = k - m, j = m$.

The original domain is a rectangle: $0 \le i \le M, 0 \le j \le N$. Substituting the inverse mapping yields constraints on the new variables: $0 \le k - m \le M, 0 \le m \le N$. These constraints define a parallelogram in (k, m) space, and the compiler must express it as nested loop bounds in loop order. For $\mathit{loopOrd}(k, m)$, the compiler first solves for the outer variable k by eliminating m, then solves for the inner variable m given k.

Shearing is valid when two conditions hold. First, the coordinate map must be invertible over the integers. The compiler builds the map from the user's *shear* arguments and computes its determinant symbolically with SymPy [51], admitting the map only when the determinant is ±1 (unimodular), which affirms the map is invertible. Second, under the chosen loop order, every dependency of the rewritten recurrence must point to a cell visited earlier. The compiler substitutes the new coordinates into the recurrence and re-runs the dependency analysis. For $\mathit{shear}(k = \mathit{antidiag}(i, j), m = j)$, the sheared edit distance recurrence depends on $F_{k-2,m-1}$, $F_{k-1,m}$, and $F_{k-1,m-1}$, which all lie on earlier antidiagonals $k - 2$ and $k - 1$. Under $\mathit{loopOrd}(k, m)$, these cells are computed before $F_{k,m}$, so the loop order is legal. If the chosen loop order placed any dependency after the cell that reads it, the compiler rejects the program (Section 6.1).

Loop reordering and shearing both reorganize how cells are visited and stored, but they still visit every cell in the matrix. For edit distance between two sequences of lengths (M) and (N), this means filling all $(M \times N)$ entries, even when the sequences are nearly identical and the answer is a small number of edits compared to the matrix length. This waste compounds at scale. Production pipelines pick regions within a genome and run the DP aligner only on those, each up to tens of thousands of base pairs [27]. Within such a segment, the edit distance is typically small relative to the segment dimensions: a segment of length (10,000), for example, may contain only a few dozen edits. The optimal alignment therefore occupies only a small fraction of the DP matrix. Thus, the vast majority of cells contain large edit distances that play no role in the optimal alignment, making their computation unnecessary.

As illustrated in Figure 9, there is a more natural way to think about traversal, described in the literature [48, 54, 76]. Instead of sweeping through positions and computing costs, the iteration

can sweep through costs and discover positions. The search starts with zero edits and sees how far it gets. Then it allows one edit and sees where that takes it, then two edits, and so on. Once the search has reached the end of the matrix, it has found the edit distance (which is very small for similar sequences like the sequences in Figure 9).

6.3 Value-Based Iteration through Search

The transformation applies to recurrences that find an optimal alignment or path by minimizing or maximizing a score, because these are fundamentally shortest-path problems: each possible alignment corresponds to a path through the DP matrix. Search requires that the score changes monotonically along every path. Each dependency must leave the score unchanged or move it in a single direction, allowing search to sweep scores in order. Therefore, the recurrence can be reformulated to find the path with the lowest (or highest) total cost to the endpoint. For example, as described in Section 9.3, Needleman Wunsch and affine gap alignment both search for an optimal alignment by choosing, at each cell of the DP matrix, the best among matching, mismatching, inserting, or deleting. This same structure holds for many biology problems and in particular applies to recurrences whose aggregator is a min or max, and in which all costly transitions have strictly positive weights (for min) or strictly negative weights (for max), ensuring that scores change monotonically along any path through the matrix.

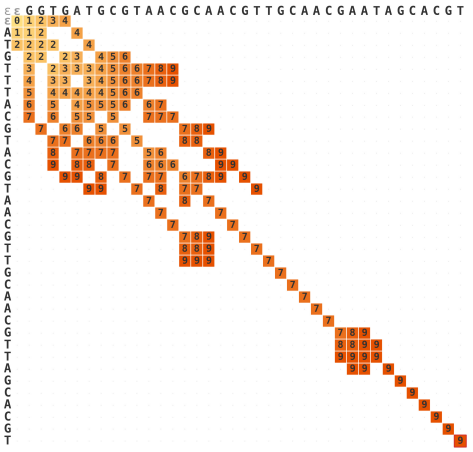


Fig. 9. Search iteration for edit distance. Each cell is colored by the number of edits needed to reach it.

Language Feature. The search transformation implements this by inverting the roles of index and value in the recurrence. The standard edit distance recurrence indexes by position—cell (i, j) stores the edit distance between the first i characters of the query and the first j characters of the reference. The user requests the inversion through the *search* command: for example, *search*(*score*, *d*=*diag*(i, j)).

The keyword *score* tells the compiler to use the value computed by the original recurrence—i.e. the edit distance—as the new outer index. Instead of sweeping through all positions and discovering costs, the transformed recurrence sweeps through costs and discovers positions. The remaining coordinate, *diag*(i, j), organizes (e.g., indexes) cells within each score level.

The *score* argument is fixed—it is always the value of the original recurrence—but the second argument is a user-specified index variable. The constraint is that transitions of *cost zero* must preserve whatever indexing variable is chosen. For edit distance, a character match advances both i and j by one. The transition from $F_{i-1,j-1}$ to $F_{i,j}$ does not incur a scoring cost during a match (as per the edit distance recurrence), and the diagonal $d = j - i$ is unchanged. Therefore, *diag*(i, j) satisfies the constraint. The compiler can verify the constraint automatically: it checks that every zero-cost transition in the original recurrence leaves the chosen index variable unchanged.

The derived recurrence is a non-obvious transformation; the intuition for why this works, and how the compiler derives the new recurrence, are described over the course of this section. The result of this transformation is that for similar sequences the algorithm terminates as soon as it

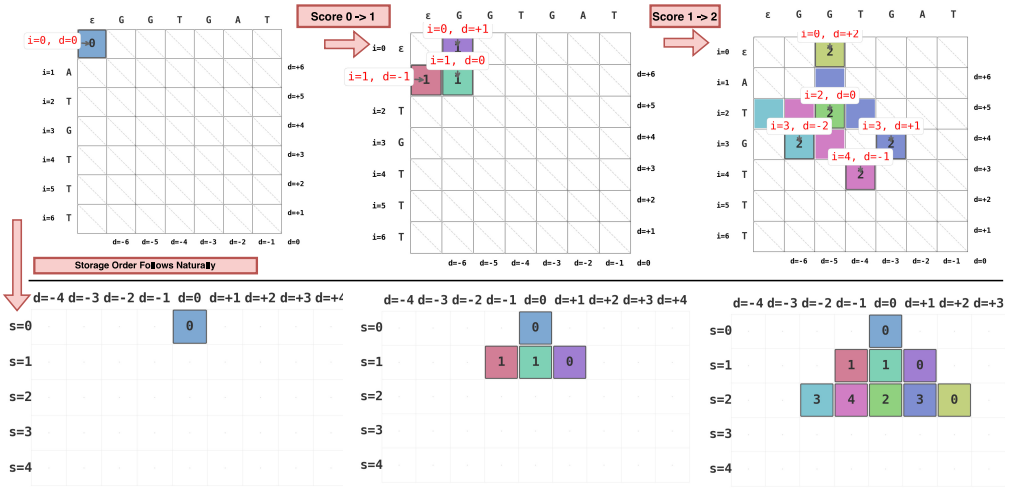


Fig. 10. Step-by-step search expansion. Each panel shows one score level's frontier over the original (i, j) matrix (top) and the corresponding row in the cost-indexed recurrence (bottom). The color of each cell on the top corresponds with the color of the cell on the bottom. Notice that the upper matrices store score and the lower ones store indices.

reaches the end of both sequences and only explores cells with less cost than the final edit distance as seen in Figure 9.

RIR Rewrite. The transformation hinges on a distinction that some transitions are *free* and some are *costly*. In edit distance, a diagonal step where the characters match costs nothing—the alignment advances in both sequences without spending an edit. The other transitions—substitution (mismatch), insertion, deletion—each cost exactly one. Any recurrence amenable to this transformation has the same two-part structure.

This partition maps naturally onto a graph perspective. The alignment matrix is analogous to a directed graph in which each cell is a node. Costly transitions—insertions, deletions, mismatches—are weighted edges (cost 1). Free transitions—character matches—are zero-weight edges. The search transformation is then a form of Dijkstra-like [16] exploration on this graph: it processes nodes in order of cost, expanding outward from the origin one score level at a time until it reaches the endpoint (endpoint is the bottom-right corner of the original matrix).

But the structure of the edit distance graph allows something more efficient than generic shortest-path search. Free transitions have a nice geometric property: they preserve the $diag(i, j)$ coordinate. A match advances both i and j by one, so $d = j - i$ is the same before and after. Furthermore, the score does not change during a match. In the edit distance equation, this can be observed because the diagonal transition is defined by $F_{i,j} = F_{i-1,j-1} + [q_i \neq r_j]$. During a match, $q_i = r_j$, meaning $F_{i,j} = F_{i-1,j-1}$. This means a run of consecutive matches stays on a single diagonal and preserves the score, so the entire run can be collapsed into one number—the furthest row reached. For example in Figure 9, the long run of 7s and 9s along diagonal 0 corresponds to a run of matches.

Costly transitions, by contrast, shift the diagonal by a fixed amount. A deletion steps right (j increases), shifting to diagonal $d + 1$. An insertion steps down (i increases), shifting to diagonal $d - 1$. A mismatch steps diagonally on a non-matching pair, staying on diagonal d. The complete set of diagonal shifts is $\{-1, 0, +1\}$, and each shift contributes one term to the three-way max in the transformed recurrence.

Figure 10 walks through the transformation on a concrete sequence pair. Each panel shows one score level's expansion over the original matrix (top) and the corresponding row in the cost-indexed recurrence (bottom). The storage order (bottom) follows naturally from the iteration pattern (top).

Score 0. The search begins at a single node: the top-left corner of the alignment matrix, sitting on diagonal 0 at row 0.

Score 1. From every position on the score-0 wavefront, the iteration asks "what are the neighboring cells or edges that can be reached by spending exactly one edit?" There are three kinds of moves—a substitution (step diagonally, staying on the same diagonal d), an insertion (step down, shifting to diagonal $d - 1$), or a deletion (step right, shifting to diagonal $d + 1$).

Score 2. The iteration repeats: each score 1 point generates three edit neighbors, reaching new diagonals at score 2. When two paths arrive at the same diagonal, only the one that reaches further survives—the other is dropped, since it spent the same number of edits but made less progress towards the endpoint. After creating this initial set of neighbors, each surviving DP cell can move forward through consecutive character matches for free. Next, the points on diagonals -2, -1, and 1 each advance through matching characters (G=G, GT=GT, and TG=TG) without spending an edit. The purple point at ($d = +1$, score 1) is an illustrative example. It fans out to diagonals 0, $+1$, and -1 at score 2 (Figure 10). The move to $d - 1$ is dropped—that diagonal was already reached at score 1 with fewer edits; however, the moves to d and $d + 1$ continue. Move d can advance through the two matching characters (TG=TG).

The storage layout and the recurrence definition follow naturally from the search intuition. At each score level, the iteration strategy only touches a small set of diagonals—three at score 1, at most five at score 2, and so on. The iteration never revisits a lower score. So the optimized storage can be just an array per score level, indexed by diagonal, where each entry holds one number: the furthest row reached, since the goal of the algorithm is to reach the last (row, col). In the original recurrence, the algorithm terminates after filling all $M \times N$ entries; the answer sits in the bottom-right corner $F_{M,N}$. In the transformed recurrence, it terminates as soon as some score s reaches that same corner—i.e., when $F_{s,N-M} \geq M$. That score is the edit distance.

RIR Before:

$$F_{i,j} = \min\big(F_{i-1,j-1} + [\mathrm{q}_i \neq \mathrm{r}_j],\ F_{i-1,j} + 1,\ F_{i,j-1} + 1\big)$$
$$0 \leq i \leq M, \qquad 0 \leq j \leq N$$

RIR After:

$$F_{s,d} = \max\big(F_{s-1,d} + 1,\ F_{s-1,d+1} + 1,\ F_{s-1,d-1}\big) + \mathrm{extend}(s, d)$$
$$0 \leq s \leq M + N, \qquad \mathrm{diag_min}_s \leq d \leq \mathrm{diag_max}_s$$
$$\mathrm{diag_min}_s = \mathrm{diag_min}_{s-1} - 1, \qquad \mathrm{diag_max}_s = \mathrm{diag_max}_{s-1} + 1$$

The indices are now score s and diagonal d instead of positions i and j. The min has become a max: the original recurrence minimizes cost, while the new one maximizes position (the furthest row reached on each diagonal). The three terms inside the max correspond to the three edit operations—mismatch, insertion, deletion—but expressed as shifts between diagonals rather than steps between cells. An extend function follows the max, greedily advancing through consecutive character matches for free. And the domain is no longer a fixed rectangle; it expands at each score level, with recursive bounds that widen by one diagonal in each direction per score.

Two features of the RIR make this expressible. First, the RIR allows domain bounds that depend on the previous level (domain recurrences) and are computed by auxiliary recurrences, so the expanding diagonal range can be represented directly as recursive bounds. Second, the RIR separates recurrence expressions from domain constraints, allowing the compiler to replace the position-indexed recurrence with a cost-indexed one while independently deriving the expanding domain.

The partition into free and costly transitions of the original recurrence determines the structure. Every free transition—those with zero cost in the original recurrence—goes into the extend function, since these are the moves that can be followed greedily without increasing the score.

The transitions determine the terms of the new recurrence. Each such transition shifts the non-score coordinate by a fixed amount; the compiler collects these into a shift set. For edit distance, the shifts are $\{-1, 0, +1\}$, producing three terms in the max. A recurrence with different edit operations—say, one that also allowed swapping of adjacent characters—would have a different shift set and a correspondingly different number of terms.

The shift set also determines the expanding domain. At score 0, only the origin diagonal is active. Each transition can shift the diagonal by at most 1, so the range of active diagonals widens by at most 1 in each direction per score level—the recursive domain bounds in the RIR. A recurrence whose transitions could shift by up to k would see its domain widen by k per level instead.

Finally, the termination condition follows from the coordinate mapping: the original recurrence terminates when it fills cell (M, N); the transformed recurrence terminates when diagonal $N - M$ reaches row M, which is the same cell expressed in the new coordinates.

Search safely applies to a recurrence provided four conditions hold. These conditions rely on the recurrence's transition costs—the change in score when moving from neighboring, dependent cells to the current cell. First, the recurrence must aggregate over neighboring cells' integer-valued scores with a min or max expression. Second, transitions must change the score monotonically: for min the score may only increase or stay equal, and for max it may only decrease or stay equal. Third, a zero-cost transition must leave the second argument passed to *search* unchanged. Fourth, score-changing transitions must shift that secondary coordinate by a constant.

7 Pruning Language

In two-dimensional sequence alignment, each dynamic programming recurrence corresponds to an alignment matrix. Matrix cells near the main diagonal represent alignments with relatively few insertions or deletions, while cells far from the diagonal correspond to large deviations between the reference and query sequences. When two sequences are expected to be similar, exploring far-off-diagonal cells is both unnecessary and wasteful. This reflects a basic biological intuition: mutations tend to be rare and spread out, and alignment recurrences should focus computation on regions where the sequences align well. More generally, many bioinformatics recurrences have large regions of the matrix that are unlikely to contribute to the final answer.

```
PruneProg ::= PruneConstruct*
PruneConstruct ::= PruneAxis | Stmt   (from Figure 5)
PruneAxis ::= Expr CmpOp CoordExpr CmpOp Expr
CoordExpr ::= diag(v, v) | antidiag(v, v) | v| | ...
```

Fig. 11. Grammar of the Pruning Language. Pruning language is syntactically RIR statements but written in user coordinates and compiled into loop order coordinates by the compiler.

Pruning restricts which parts of the matrix get computed, skipping these regions entirely. Because pruning discards cells that the original recurrence computes based on heuristics, it is a non-semantics-preserving transformation (final recurrence values and traceback path may be different to the unpruned, unscheduled version). The user is therefore responsible for ensuring—based on biological intuition—that the omitted regions do not significantly affect the final result. The grammar for Filtr's pruning language is given in Figure 11.

The pruning language provides a language feature—the *prune axis*—which is an inequality that restricts iteration to a geometric sub-region of the matrix. For example, a band around the main diagonal is denoted by *prune axis*: $-B \leq \text{diag}(i, j) \leq B$. The user writes an inequality that bounds a coordinate expression (e.g., $\text{diag}(i, j)$) over the loop indices, and only cells satisfying that inequality are ever visited. The coordinate expression can be any function of the indices that is invertible, but generally is just $\text{diag}(i, j)$ or $\textit{antidiag}(i, j)$. The bounds of a prune axis may also be recurrence

expressions rather than constants, allowing the geometric region to evolve dynamically at runtime. Therefore, a pruning program may be either *static*, with constant bounds, or *dynamic*, where the user can specify domain recurrences that update the prune axis bounds at runtime.

The prune axis requires the coordinate expression (e.g., $\text{diag}(i, j)$) to be a linear combination of the loop indices. This linearity allows the compiler to automatically calculate the new loop bounds and guarantees that the remaining valid cells always form a single, contiguous band. When the bounds are recurrences (dynamic pruning), each bound governing a frontier may read only cells computed on strictly earlier frontiers.

The Pruning Language serves two purposes: (1) it separates concerns (pruning, iteration, and base recurrence) for the user, and (2) it allows for a flexible mixing of coordinate systems. Figure 12 shows k and m from the antidiagonal coordinate system being used in tandem with i and j. Specifically, the recurrences should be indexed by k and m (i.e., the *loopOrd* variables) but coordinate expressions (e.g. $\text{diag}(i, j)$) can be defined in either coordinate system (e.g., $\text{diag}(i, j)$, $j - i$, or $2m - k$).

The pruning model focuses on supporting real-world heuristics but does not allow for arbitrary pruning. Specifically, it targets pruning techniques over contiguous geometric regions, assuming that at each step there is a single continuous active region and that pruning decisions rely only on information available at the current step rather than future values. In practice, we have not encountered algorithms that violate either assumption, and the language can be easily extended to support additional pruning needs in the future.

Although pruning programs are often longer than those in the Recurrence Language or the Iteration Ordering Language, each one is still small (Figure 12 shows the five-line X-drop pruning program) and can be applied to many different recurrence equations. This same X-drop pruning program, for example, can be composed with edit distance, affine gap alignment, or other recurrences as described in Section 9.

7.1 Static Pruning

In static pruning, the bounds are fixed at compile time before execution. For example, in banded edit distance, if the user knows in advance that the best alignment will stay within B diagonals of the main diagonal, then they can use static pruning. The pruning program is a single *prune axis*: $-B \le \text{diag}(i, j) \le B$.

The user can write this constraint in whichever coordinate system is most natural, as long as the expression can be algebraically rewritten in terms of the *loopOrd* variables. To illustrate, the compiler internally knows that $diag(i, j) = j - i$ and $antidiag(i, j) = i + j$. Filtr automatically converts the user-provided constraints so they are defined strictly in terms of the desired loop iteration variables (using the SymPy computer algebra system). The compiler then intersects the rewritten constraint bounds with the existing domain bounds. For column-major traversal with $loopOrd(j, i)$, the compiler solves $\text{diag}(i, j) = j - i$ for i, yielding $0 \le j \le N,\ \max(0, j-B) \le i \le \min(M, j+B)$. Under $loopOrd(k, m)$ where $k = antidiag(i, j)$ and $m = j$, the compiler algebraically solves $\text{diag}(i, j) = j - i$ for m and k, which resolves to $\text{diag}(i, j) = 2m - k$, yielding $0 \le k \le M + N,\quad \max\big(0,\ k - M,\ \lceil (k - B)/2 \rceil\big) \le m \le \min\big(N,\ k,\ \lfloor (k + B)/2 \rfloor\big)$. Under a *search(score, d=diag(i,j))* transformation, the constraint maps to $-B \le d \le B$ in the (s, d) loop space.

7.2 Dynamic Pruning

To compute dynamic prune axis bounds at runtime, the Pruning Language provides three additional constructs—recurrence equations, conditional recurrences, and set-builder reductions—that are syntactically a subset of the RIR but written with respect to the Recurrence Language input. These equations allow the user to describe the domain as a recurrence (domain recurrence), track

auxiliary information such as a running minimum score, and conditionally override a cell's value (e.g., replacing it with ∞ when a pruning condition is met).

In *dynamic pruning*, the boundaries adapt at runtime based on computed values. This requires all the constructs working together, as shown in X-drop in Figure 12; an auxiliary recurrence analyzes the most recently computed cells ($min_score_k =$); a conditional recurrence uses this analysis to mark cells for removal ($F_{k,m} = \infty$ if); and set-builder reductions calculate the new bounds based on these markings ($\min(\text{diag}(i, j))$ if $F_{k,m} \neq \infty$). The result is a feedback loop—what is computed (data recurrence) determines where computation happens next (domain recurrences).

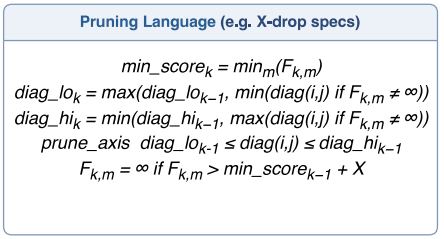


Fig. 12. Pruning specification for X-drop.

To ensure this feedback loop executes safely, dynamic pruning must meet two preconditions. First, the pruning bound area must be defined by one upper and one lower linear bound. Second, current bounds can only be calculated using data from completed previous iterations.

Other pruning programs follow a similar structure. Z-drop, for instance, replaces the X-drop conditional with one that measures distance in addition to score and is described in Section 9.3. Similarly, after applying a search transformation to a recurrence, the user can derive a pruned version of the resulting search algorithm, such as WFA-Adapt, which prunes diagonals based on distance from the diagonal closest to the target endpoint (also described in Section 9.3).

The compiler lowers this pruning program into the Recurrence IR by converting all expressions into the loop order coordinate scheme, introducing min_score_k as a recurrence, splitting the main recurrence into conditional regions, and substituting the domain recurrences for static loop bounds. The resulting RIR is the X-drop formulation shown in Figure 6 (right). The domain bounds diag_lo_k depend on $F_{k,m}$, while the iteration of $F_{k,m}$ is controlled by those same bounds at the previous timestep—this bidirectional dependence between what is computed and where computation happens is what distinguishes dynamic pruning from static banding.

8 Code Generation

The Filtr compiler transforms the rewritten RIR into C++ code. We adopt the general approach of the Recuma recurrence compiler [71], which constructs a dependency graph from recurrence definitions and lowers it into loop code. In our case, the RIR serves as the input to this lowering. Figure 13 shows the complete pipeline for X-drop edit distance: the user-facing Filtr input (left), the generated C++ (center), and the code generation algorithm (right).

The Filtr compiler first applies iteration rewrites and then applies pruning rewrites. Applying iteration rewrites first allows the compiler to verify that the transformed traversal order respects the recurrence's data dependencies against the full, unpruned recurrence. Applying pruning second is necessary because dynamic pruning strategies like X-drop decide what to prune based on what has been computed so far, meaning their behavior depends on the order in which states are visited. As such, pruning recurrences must be expressed in the coordinate system produced by the iteration rewrites. For example, an X-drop specification references antidiagonal indices, which only exist after the shearing transformation has been applied.

The main challenge of lowering the RIR to C++ is the dependency between data and domain recurrences. Domain variables such as diag_lo and diag_hi are defined by recurrences that read from the main DP array, yet those same variables determine the loop bounds over which the DP array is computed. Filtr resolves this through staged code generation within a single loop body, organized into three stages. As shown in the codegen algorithm in Figure 13 (right), the compiler

Python Filtr input (for X-drop)

```
recurrence = (F[i,j]=min(F[i-1, j-1] + [q[i] != r[j]],
                         F[i-1, j] + 1,
                         F[i, j-1] + 1)
              F[0,0]=0, F[i,0]=i, F[0,j]=j)

iteration = (shear(k = antidiag(i,j), m = j), loopOrd(k, m))

pruning = (
  min_score[k] = min(m, F[k, m] ),
  F[k,m] = INF if F[k,m] > min_score[k-1] + X,
  diag_lo[k] = max(diag_lo[k-1],
                   min(diag(i,j) if F[k,m]!=INF )),
  diag_hi[k] = min(diag_hi[k-1],
                   max(diag(i,j) if F[k,m]!=INF )),
  prune_axis = diag_lo[k-1] <= diag(i,j) <= diag_hi[k-1]
)

Filtr.compile(recurrence, iteration, pruning)
```

C++ Code (for X-drop)

```
F[0][0] = 0;
for (int k = 1; k <= M; ++k) {F[k][0] = k;}
for (int k = 1; k <= N; ++k) {F[k][k] = k;}
diag_lo[0] = -M; diag_hi[0] = N; min_score[0] = 0;
for (int k = 1; k <= M + N; ++k) {
  // Stage A: read prune-axis bounds from previous antidiag
  int m_lo = max({0, k - M, ceil_div2(k + diag_lo[0])});
  int m_hi = min({N, k, floor_div2(k + diag_hi[0])});
  if (m_lo > m_hi) break;
  // Stage B: compute recurrence + apply pruning
  min_score[1] = INF;
  for (int m = m_lo; m <= m_hi; ++m) {
      int val = min({F[k-2][m-1] + (q[k-m] != r[m]),
                     F[k-1][m]   + 1,F[k-1][m-1] + 1});
      F[k][m] = (val > min_score[0] + X) ? INF : val;
      min_score[1] = min(min_score[1], F[k][m]);}}
  if (min_score[1] == INF) break;
  // Stage C: tighten surviving diagonal band for next round
  diag_lo[1] = diag_lo[0];
  diag_hi[1] = diag_hi[0];
  for (int m = m_lo; m <= m_hi; ++m) {
      if (F[k][m] != INF){diag_lo[1] = max(diag_lo[0], 2*m-k);
                          break;}}
  for (int m = m_hi; m >= m_lo; --m) {
      if (F[k][m] != INF) {diag_hi[1] = min(diag_hi[0], 2*m-k);
                           break;}}
  diag_lo[0] = diag_lo[1];
  diag_hi[0] = diag_hi[1];
  min_score[0] = min_score[1];}
```

```
def codegen(rir):
  # 1. For each equation, collect which arrays its RHS references.
  #   e.g., F[k,m] reads F[k-2,m-1], F[k-1,m], F[k-1,m-1]
  #   e.g., min_score[k] reads F[k,m]; diag_lo[k] reads diag_lo[k-1] and F[k,m]
  deps = {eq.lhs: extract_array_refs(eq.rhs) for eq in rir}
  # 2. Classify each equation by its role in the recurrence:
  #   DATA:   main DP arrays (F, or M/I/D in affine gap)
  #   DOMAIN: arrays that appear in any loop bound (diag_lo, diag_hi)
  kind = classify(deps, rir)
  # 3. Find mutually-recursive eqn groups via SCC, then topo-sort.
  groups = scc(eqns where kind != DOMAIN, deps)
  body_order = topo_sort(groups)
  # 4. Walk the RIR's domain -- emit nested loops.
  for depth, (var, lo, hi) in enumerate(RIR.domain):
      # 5. If any DOMAIN array feeds into bounds, read from prev step.
      #   e.g., at depth 0 (k-loop): read diag_lo[k-1], diag_hi[k-1]
      domain_deps = [eq for eq in kind[DOMAIN] if var in eq.bound_vars]
      emit_bounds_read(domain_deps, var)
      emit_loop(var, lo, hi)
      # 7. Innermost body: emit equation groups in topological order.
      #   Each is a single eqn or a jointly-computed SCC (e.g., M,I,D at one cell).
      for group in body_order:
          emit_cell(group)
      # 8. Close loops inside-out.
      for depth in reversed(range(len(loops))):
          emit_end_loop()
          # 9. Update DOMAIN recurrence indexed by this level.
          domain_updates = [eq for eq in kind[DOMAIN]
                            if loops[depth].var in eq.index_vars]
```

Fig. 13. End-to-end example of Filtr compiling X-drop edit distance, showing the user-facing input specification (left), simplified generated C++ (center), and the code generation algorithm (right). The C++ code is generated code slightly cleaned for readability.

first classifies each equation by its role: *data* recurrences are the main DP arrays (e.g., F, or M, I, D in affine gap), and *domain* recurrences are arrays that appear in loop bounds (e.g., diag_lo, diag_hi). It then topologically sorts the non-domain equations to determine the order in which they are emitted. A domain recurrence is emitted immediately after all of its data dependencies are satisfied (i.e., after the respective data recurrences that it depends on have been emitted).

The final output arranges arrays according to the iteration order's indexing scheme: for antidiagonal traversal, arrays are indexed by antidiagonal number and offset; for score-based indexing, by score layer and diagonal. Filtr relies on vectorization hints and autovectorization to generate performant C++ code.

9 Evaluation

Our evaluation supports three claims: (1) Filtr-generated code matches or exceeds the performance of hand-optimized bioinformatics libraries and existing recurrence compilers; (2) Filtr's composable interface lets users explore combinations of pruning and traversal strategies to find the best configuration for a given dataset; and (3) pruning specifications written for one recurrence transfer directly to others, making it easy to construct new heuristics.

9.1 Experimental Methodology

There are four broad classes of bioinformatics algorithms that rely on recurrences: sequence alignment, RNA folding (predicting how an RNA molecule folds into a three-dimensional structure), sequence prediction models (used to identify patterns such as genes within a sequence), and sequence chaining (linking short matching alignments into a longer alignment). There is a diverse set of recurrences used in the evaluations, but they generally resemble the examples in Table 1. Needleman Wunsch extends edit distance by allowing arbitrary substitution and gap scores. Affine-gap alignment is the three-state sequence-alignment recurrence shown in Table 1. Nussinov is the RNA-folding recurrence shown in Table 1. Viterbi is the sequence-prediction recurrence shown in Table 1. Zuker minimum-free-energy folding is similar to Nussinov, but it minimizes energy rather than maximizing the number of base pairs. McCaskill sums contributions from many possible structures instead of choosing among a fixed set of four cases. Although these recurrences have

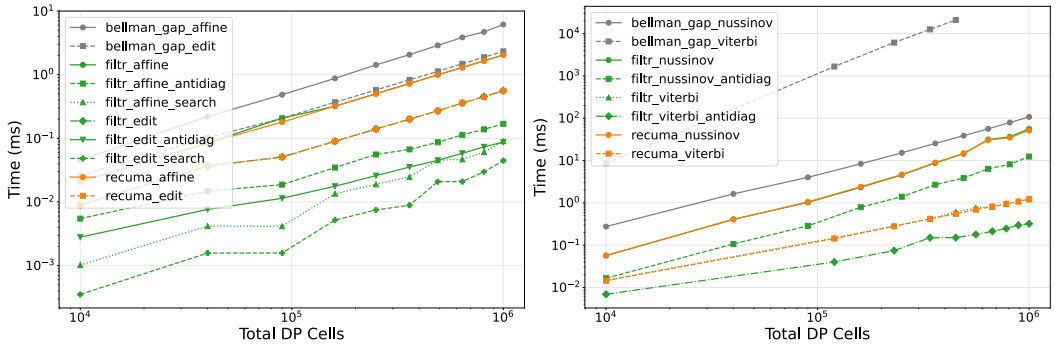


Fig. 14. Filtr versus Recuma and Bellman Gap. Runtime (log-log) across four recurrences: edit distance, affine gap penalty (left), Nussinov RNA folding, and Viterbi sequence prediction (right).

more complicated dependency structures than edit distance, we can still apply the same diagonal, antidiagonal, and search transformations.

All experiments were conducted single-threaded on an Intel Core i9-285K machine with 64 GiB RAM using GCC 13.3.0 (Ubuntu 24.04), CMake 3.28.3, and C++23. Filtr was compiled with `-O3 -march=native -mavx2 -flto -funroll-loops -ffast-math`. The two highest and two lowest measurements were discarded, and the reported runtime is the mean of the remaining six runs.

We report runtime in milliseconds and, where applicable, alignment accuracy as the percentage difference between exact and approximate results. Library and compiler comparisons use synthetic sequences so that divergence level, sequence length, and mutation profile can be controlled independently, isolating performance differences. The evaluation of design-space exploration (Section 9.4) and heuristic transfer (Section 9.5) uses a wide variety of real DNA and RNA sequences to assess how Filtr behaves under authentic biological conditions.

9.2 Filtr vs. Recurrence Compilers

Recuma, the primary compiler baseline, represents the current state of the art in recurrence compilation. It generates efficient code for standard row-by-row and column-by-column traversals of the DP matrix but lacks support for alternative traversal orders, pruning strategies, or score-indexed traversal. Our evaluation datasets include synthetic query–reference sequence pairs that are 90% similar and representative of many real genomic datasets [33, 86].

Figure 14 plots the runtime against the problem size for different configurations across four recurrences: edit distance, affine gap penalty, Nussinov [57] (RNA folding), and Viterbi (sequence prediction). Unoptimized Filtr serves as an ablation baseline. On a standard row-wise traversal, Filtr matches Recuma's runtime closely at all tested sizes, confirming that Filtr's code generation introduces no overhead relative to an existing recurrence compiler. Antidiagonal iteration, which exposes automatic SIMD parallelism, yields a 7–13× speedup over Recuma and unoptimized Filtr at medium to large sizes for affine gap penalty, and a 6–8× speedup for edit distance on asymmetric problems. With the search transformation applied, edit distance achieves 14–20× speedups and affine gap achieves 25–50×. However, as we show in Section 9.4, these gains depend on the input data. For example, search performs best on similar sequences and its advantage diminishes as divergence increases. We also compare against Bellman's Gap, which performs comparably to Recuma; however, it is similarly slower than Filtr kernels.

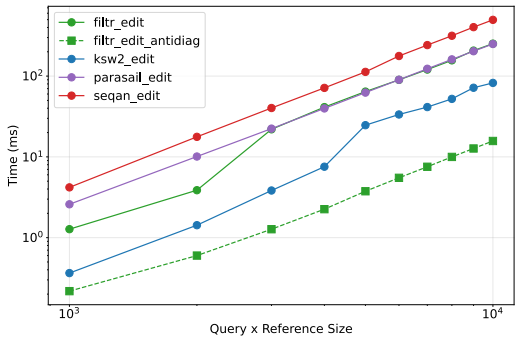


Fig. 15. Filtr versus hand-optimized unpruned alignment libraries (Ksw2, Parasail, SeqAn) on edit distance. Log scale.

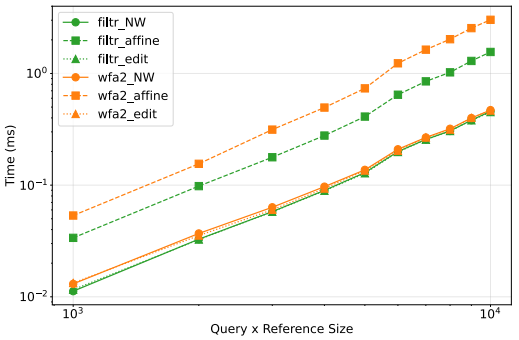


Fig. 16. Filtr search vs. WFA2. We can additionally apply this search to longest common subsequence, dual gap, chaining, and RNA folding. Log scale.

Tiling Compiler. We also compare Filtr against Reptile [73], a compiler that automatically tiles recurrences. Table 2 shows Reptile's performance relative to unoptimized Filtr. Reptile provides modest improvements for Smith Waterman (SW) and slightly slows down Needleman Wunsch (NW). Sequence alignment kernels are primarily compute-bound rather than memory-bound, leaving little opportunity for cache tiling to improve performance. In contrast, optimized Filtr achieves 2.5× to 30× speedups over unoptimized Filtr, with shear delivering a 6× speedup. Combining Filtr with cache-aware tiling is a promising direction for future work, particularly on GPUs and other architectures.

Table 2. Reptile speedup over unoptimized Filtr using the best tile size, for sequence lengths 64–1024.

	64	320	512	768	1024
NW	1.09×	0.90×	0.88×	0.88×	0.86×
SW	1.26×	1.20×	1.19×	1.17×	1.18×

9.3 Filtr vs. Hand-Optimized Libraries

Sequence alignment is one of the most performance-critical operations in bioinformatics, and the community has invested heavily in hand-tuned, library-level implementations [14, 39, 48, 61]. We compare Filtr against four widely used libraries: SeqAn [61] (which provides unpruned, banded, and X-drop alignment), Parasail [14] (which provides unpruned and banded), Ksw2 [39] (which provides a Z-drop variant), and WFA2 [48] (which provides a search variant). Parasail, WFA2, and Ksw2 are hand-vectorized implementations.

Unpruned Comparison. The Needleman Wunsch algorithm and its simpler edit distance variant use a recurrence to compute the optimal global alignment between two sequences. As shown in Figure 15, Filtr's antidiagonal shearing iteration is the fastest among the general recurrence-based implementations, outperforming baseline methods by 5–30× because its shearing transformations ensure consecutively traversed cells are contiguous in a cache-efficient memory layout.

For the restricted task of edit distance, we also compare against Edlib [69], which packs multiple DP cells into each machine word and updates them with bitwise operations. Edlib's specialized approach does not generalize to broader bioinformatics recurrences like Needleman Wunsch and affine gap, but within its supported setting it is highly optimized, running approximately 7.5× faster than Filtr's shear and 1.6× faster than search on average (Table 3). Edlib's packing optimization could be incorporated into Filtr, although it would apply only to the subset of recurrences whose neighboring cell scores differ by at most one. We leave this extension as future work.

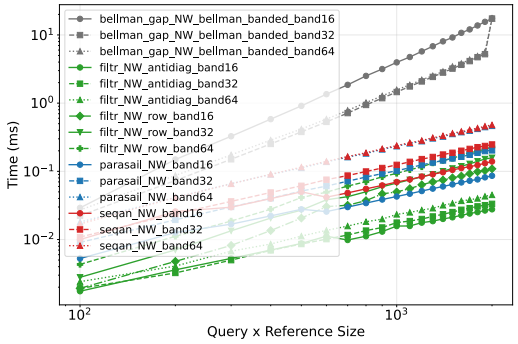


Fig. 17. Filtr vs. banded alignment libraries and compilers. Log scale.

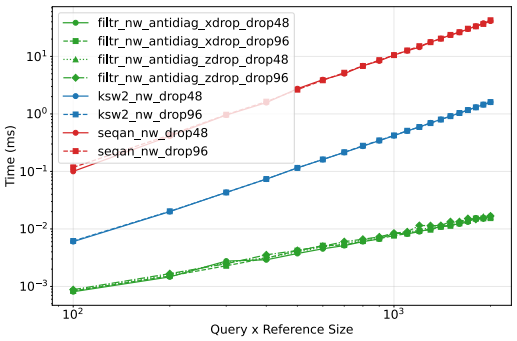


Fig. 18. Filtr vs. dynamic pruning libraries. Log scale.

We also compare Filtr's search transformation against the hand-optimized WFA2 library since this is another exact method. The search transformation corresponds to the wavefront algorithm (realized in WFA2 [48]). Figure 16 compares Filtr and WFA2 across three scoring models—Needleman Wunsch, edit distance, and affine gap penalty. Filtr remains the best. Because Filtr formulates the wavefront traversal as a standard recurrence over score coordinates, the compiler can apply automatic vectorization.

Table 3. Speedup of optimized Filtr and Edlib over unoptimized Filtr baseline.

Size	Filtr+Search	Filtr+Shear	Edlib
1000	22.3×	4.2×	10.6×
3000	16.0×	4.9×	15.6×
5000	18.7×	4.0×	34.4×
8000	17.6×	3.5×	33.9×
10000	18.6×	3.4×	47.2×

Static Pruning. Filtr's banded Needleman Wunsch kernel substantially outperforms SeqAn and Parasail across all tested band sizes (16, 32, and 64). Figure 17 shows that Filtr's antidiagonal banded kernels (green) outperform both Parasail (blue) and SeqAn (red) across all band widths and sequence lengths, often by an order of magnitude at larger sizes. Filtr also outperforms the Bellman's Gap compiler. Filtr's speedup is due to its shearing transformation over memory layouts, enabling efficient access along antidiagonals, whereas other libraries use a hard-coded row-major layout. Filtr's row-wise banded kernels (green) are competitive with or faster than the hand-tuned libraries.

Dynamic Pruning. Next, we compare heuristic methods that prune computed cells dynamically based on alignment scores rather than a fixed band based boundary. X-drop stops extending the alignment when the current score drops more than a fixed threshold below the best score seen so far. Z-drop is similar but additionally penalizes distance from the best-scoring diagonal, making it more tolerant of extended gaps and score fluctuations that arise from long insertions or deletions. In our language, the conditional recurrence in the pruning program would be:

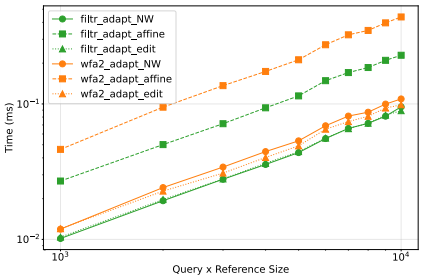


Fig. 19. Filtr vs. WFA-Adapt. Log scale.

$$F_{k,m} = -\infty \quad \text{if} \quad \text{max_score}_{k-1} - val > Z + \left|\text{diag}(i,j) - \text{diag_best}_{k-1}\right|$$

This condition corresponds to the Z-drop heuristic. Since affine gap is a max recurrence, cells that violate the Z-drop condition are invalidated by assigning them a score of $-\infty$. Here, max_score denotes the best alignment score, at coordinate diag_best_k (which is determined with argmax data recurrences). Figure 18 shows that Filtr's X-drop and Z-drop variants (green) outperform both

Ksw2 (blue) and SeqAn (red) by one to two orders of magnitude across all sequence lengths. Both Ksw2 and SeqAn traverse without shearing the storage structure. Notably, Ksw2 does not provide an X-drop algorithm, and neither SeqAn nor Parasail supports Z-drop, whereas Filtr generates both X-drop and Z-drop from the same recurrence specification with different pruning conditions.

We also compare against dynamic pruning strategies applied to search. WFA-Adapt[48] employs a heuristic that, at each score level, discards diagonals falling too far from the leading diagonal (the diagonal representing the most progress through the alignment matrix). Filtr can express this data-dependent pruning, achieving performance comparable to this library (Figure 19).

Table 4. Search traversal (SR) speedup over antidiagonal traversal (AD) for edit distance.

Div. %	Size	SR (ms)	AD (ms)	Speedup
1	500×500	0.7	69	97×
1	2000×2000	8.9	870	98×
1	4000×4000	13.8	4150	300×
5	500×500	1.9	50	27×
5	2000×2000	52	710	14×
5	4000×4000	190	4270	23×
30	500×500	110	51	0.45×
30	2000×2000	1580	720	0.45×
30	4000×4000	6020	4360	0.72×

9.4 Exploring the Design Space

We have established that Filtr matches or exceeds existing systems on individual benchmarks. Now, we demonstrate that the best heuristic and schedule depend on properties of the input data. This means there is no single best configuration—users need the ability to explore alternative heuristics.

Search vs. Antidiagonal Traversal. Table 4 reports the speedup of search-based (score-indexed) traversal over antidiagonal traversal for edit distance at three divergence levels and three matrix sizes. At 1% divergence, the search transformation is 97–300× faster because it searches only the narrow strip of cells at each score level, while antidiagonal traversal fills the entire matrix. At 5% divergence, the advantage narrows to 14–27×, and at 30% divergence, search is slower (0.45–0.72×) because the search expands to cover most of the matrix, and the overhead of search traversal outweighs the savings from skipping cells. The optimal strategy depends on the divergence characteristics of the input sequences. In Filtr, users can explore different scheduling transformations to choose the best schedule for their data.

Pruning across Different Sequences. The choice of pruning heuristic is equally data-dependent. We evaluated three biological conditions with different mutation profiles. For each condition, we computed the exact optimal alignment using full affine gap scoring and compared it against three faster heuristics—banded, X-drop, and Z-drop—across a sweep of parameters, measuring both runtime and accuracy relative to the ground truth.

Figure 20 presents the resulting best option for each condition. In the resequencing case (left panel, Human vs. Mouse Cytochrome c CDS), narrow banded alignment achieves near-perfect accuracy at the lowest runtime, since mutations are small and the optimal path stays close to the main diagonal. For ortholog comparisons (center panel, Human vs. Mouse mt-16S rRNA), X-drop provides the best speed–accuracy tradeoff because its data-dependent band can widen to accommodate short bursts of insertion-deletions and then contract again when the alignment returns to a well-matching region. In the structural variation case (right panel, ACE Intron 16 Alu Insertion vs. Deletion), Z-drop is the only heuristic that maintains full accuracy even in the presence of long insertions, because it is more tolerant to extended gaps. With Filtr, users can easily plug in different pruning specifications and determine which pruning is best for their data.

9.5 Transferring Heuristics

Filtr enables users to construct new heuristics by recombining existing building blocks across problem domains. To illustrate this, we applied the X-drop pruning heuristic to three RNA secondary structure prediction recurrences. This experiment is not a rigorous evaluation of X-drop for RNA

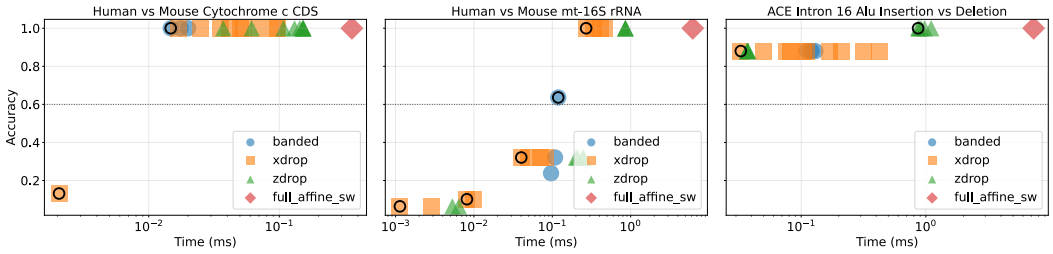


Fig. 20. Accuracy vs. runtime for three biological conditions: resequencing (Human vs. Mouse Cytochrome c CDS, left), ortholog comparison (Human vs. Mouse mt-16S rRNA, center), and structural variation (ACE Intron 16 Alu Insertion vs. Deletion, right). The optimal heuristic differs by condition: banded is useful for resequencing, X-drop for ortholog, and Z-drop for structural variation. Each dot represents a different combination of bandwidth, X-drop, and Z-drop parameters; circles mark points on the Pareto frontier. Different applications have different accuracy requirements; the dotted line at 60% marks a useful cutoff.

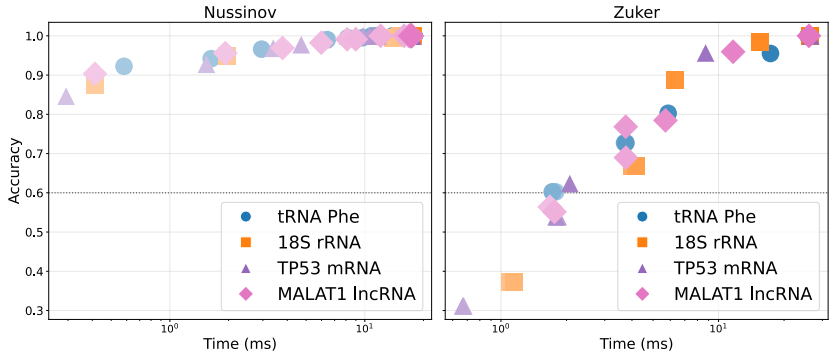


Fig. 21. Each subplot shows a recurrence evaluated on four RNA sequences. Points represent different X-drop thresholds—Nussinov: 1–400 and unpruned; Zuker: 1–1000 and unpruned. Darker shades indicate larger X-drop values (less aggressive pruning). The dotted line marks an accuracy threshold of 0.6.

folding, but a demonstration of how Filtr's composable design makes it easy to transfer a pruning strategy to a new domain.

The Nussinov recurrence [57] fills a triangular dynamic programming matrix indexed by subsequence endpoints (i, j), iterated along increasing diagonals (Table 1). We reused the same four pruning constructs from alignment-based X-drop. We additionally modified the pruning program to include a warm-up period of $\text{diag}(i, j) \leq 15$ (implemented with a conditional recurrence) during which pruning is disabled, since folding scores require several diagonals to become meaningful—unlike alignment, where even the first iterations can produce nonzero scores. We extended this transfer to two additional RNA folding recurrences—Zuker [90] (minimum free energy folding) and McCaskill [50] (partition function folding). The effectiveness of the transfer depends on the recurrence. The transfer to the McCaskill recurrence proved ineffective, likely because the recurrence aggregates contributions through multiplication and summation of exponentials rather than min or max.

Nussinov and Zuker proved far more promising. Figure 21 reports runtime and accuracy across four RNA classes. For Nussinov, X-drop is very effective: even at an aggressive threshold of $X = 2$, accuracy remains perfect across all classes; pruning removes only cells that do not contribute to the optimal fold. This is likely because Nussinov scores grow monotonically and stay close to the optimum along the main antidiagonal. For Zuker, a tight $X = 8$ threshold prunes too aggressively

(accuracy 0.3–0.6), but a larger $X = 600$ threshold restores accuracy to 0.8-0.9 while still providing modest speedup.

10 Related Work

Hand-optimized libraries like Ksw2 (used in minimap2 [39]), Parasail [14], and SeqAn [61] represent the dominant approach to achieving performance in biological sequence alignment. These systems implement recurrences directly in low-level code, embedding both the scoring semantics and the scheduling strategy within vectorized inner loops. Variants such as affine gap [28], banded [76], X-drop [88], and Z-drop [39] are typically realized as separate pieces of code, each hand-engineered for a specific heuristic. Hardware acceleration efforts, including GPU-based implementations [43] and FPGA-based efforts such as Darwin [36, 75, 80, 84] push this specialization further by synthesizing custom pipelines for alignment, achieving high throughput but sacrificing flexibility. Filtr abstracts these implementations by representing scoring and pruning as schedulable constructs.

Existing compilers for bioinformatics provide higher-level abstractions but have not generalized the structure of alignment recurrences themselves. Seq [67] is a Pythonic language for genomic pipelines that compiles to efficient code, but it targets data-flow orchestration rather than optimizing dynamic programs. SARVAVID [46] adds distributed execution and workflow parallelism but still treats alignment algorithms as opaque calls. Researchers have also developed programming systems for recurrence problems. Dynamite [8] compiles state-transition models into dynamic programming recurrences, but does not optimize its scheduling or pruning. The closest to our work are Recuma [71] and Bellman's Gap [65]. Recuma [71] formalized recurrence equations as a compilable intermediate representation, introducing a dependency–based scheduler that translates recurrence definitions to native code. However, Recuma assumes static domains and lacks primitives for pruning or dynamic domain restriction. Reptile [73] extends Recuma with tiling optimizations, but similarly does not support optimizations needed for production bioinformatics algorithms. Bellman's Gap is a logic programming system focused on reducing exponential search to polynomial recurrences. It does not support diagonal iteration or dynamic pruning, although it has limited support for pruning by placing a static band around the diagonal.

Datalog researchers have separately studied efficient compilation of (mutually) recursive query definitions. Recent work compiles recursion over ordered semirings using program synthesis and algebraic rewriting to find faster evaluation strategies [1, 66, 81]. Logic-programming frontends like Dyna [21] likewise express weighted dynamic programs declaratively as recurrences. Their focus, however, is relational query optimization or declarative logic rather than dense array recurrences.

Filtr builds on the separation of algorithm and schedule popularized by Halide [60] and TACO [35], and on loop optimization work from the polyhedral model [6, 23, 24, 37]. Similar separation-of-concerns approaches appear in Lift [70], TVM [10], and Tensor Comprehensions [77]. Halide demonstrated that schedules can independently control traversal, tiling, and fusion of dense image-processing loops. TACO generalized this concept to sparse tensor formats through algebraic iteration graphs. Filtr's iteration ordering and pruning directives are analogous to Halide's scheduling primitives but operate on semantically constrained recurrence graphs, enabling the safe transformation of irregular, score-dependent dependencies.

11 Conclusion and Future Work

Filtr brings recurrence-level programmability and scheduling separation to biological recurrences, allowing heuristic variants to be expressed, composed, and compiled automatically. Natural next steps include targeting GPUs, extending to distributed execution, and formalizing the correctness conditions for rewrites. From a user perspective, an autotuner can automatically select the best configuration for a given dataset and hardware target.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable feedback, comments, and insights on improving this paper. We thank Scott Kovach, James Dong, Christophe Gyurgyik, Rubens Lacouture, Katherine Mohr, Sai Gautham Ravipati, Alexander Root, Rohan Yadav, Atharva Chougule, and Usman Tariq for their feedback on an early draft. A special thanks to Scott for helping us formalize the separation of concerns and for helping us reason about these abstractions more rigorously. This work was supported in part by the PRISM center, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and DARPA under the Machine learning and Optimization-guided Compilers for Heterogeneous Architectures (MOCHA) program (award number HR00112520038). Shiv Sundram was supported by an NSF GRFP.

Data Availability

The artifact for this paper includes the Filtr compiler [79]. The frontend is embedded in Python, and the compiler itself is a Python repository that compiles inputs and generates C++ code. The artifact also includes scripts to reproduce the benchmarks in this paper, including the corresponding C++ kernels and the frontend code to generate them.

References

[1] Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2024. Convergence of datalog over (pre-) semirings. *J. ACM* 71, 2 (2024), 1–55. doi:10.1145/3643027

[2] Quim Aguado-Puig, Max Doblas, Christos Matzoros, Antonio Espinosa, Juan Carlos Moure, Santiago Marco-Sola, and Miquel Moreto. 2023. WFA-GPU: gap-affine pairwise read-alignment using GPUs. *Bioinformatics* 39, 12 (2023), btad701. doi:10.1093/bioinformatics/btad701

[3] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. 1990. Basic local alignment search tool. *Journal of molecular biology* 215, 3 (1990), 403–410. doi:10.1016/S0022-2836(05)80360-2

[4] Tim Anderson and Travis J Wheeler. 2024. An FPGA-based hardware accelerator supporting sensitive sequence homology filtering with profile hidden Markov models: T. Anderson, TJ Wheeler. *BMC bioinformatics* 25, 1 (2024), 247. doi:10.1186/s12859-024-05879-3

[5] James Arram, Kuen Hung Tsoi, Wayne Luk, and Peiyong Jiang. 2013. Hardware acceleration of genetic sequence alignment. In *International Symposium on Applied Reconfigurable Computing*. Springer, 13–24. doi:10.1007/978-3-642-36812-7_2

[6] Utpal Banerjee. 2007. *Loop transformations for restructuring compilers: the foundations*. Springer Science & Business Media. doi:10.1007/b102311

[7] Raphael O Betschart, Alexandre Thiéry, Domingo Aguilera-Garcia, Martin Zoche, Holger Moch, Raphael Twerenbold, Tanja Zeller, Stefan Blankenberg, and Andreas Ziegler. 2022. Comparison of calling pipelines for whole genome sequencing: an empirical study demonstrating the importance of mapping and alignment. *Scientific Reports* 12, 1 (2022), 21502. doi:10.1038/s41598-022-26181-3

[8] Ewan Birney and Richard Durbin. 1997. Dynamite: a flexible code generating language for dynamic programming methods used in sequence comparison.. In *Ismb*, Vol. 5. 56–64.

[9] Kun-Mao Chao, William R Pearson, and Webb Miller. 1992. Aligning two sequences within a specified diagonal band. *Bioinformatics* 8, 5 (1992), 481–487. doi:10.1093/bioinformatics/8.5.481

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 578–594.

[11] Francesca Chiaromonte, Von Bing Yap, and Webb Miller. 2001. Scoring pairwise genomic sequence alignments. In *Biocomputing 2002*. World Scientific, 115–126. doi:10.1142/9789812799623_0012

[12] 1000 Genomes Project Consortium et al. 2015. A global reference for human genetic variation. *Nature* 526, 7571 (2015), 68. doi:10.1038/nature15393

[13] Eliot Courtney, Amitava Datta, David H Mathews, and Max Ward. 2025. memerna: Sparse RNA folding including coaxial stacking. *Journal of molecular biology* 437, 3 (2025), 168819. doi:10.1016/j.jmb.2024.168819

[14] Jeff Daily. 2016. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics* 17, 1 (2016), 81. doi:10.1186/s12859-016-0930-z

[15] Dick de Ridder, Jeroen De Ridder, and Marcel JT Reinders. 2013. Pattern recognition in bioinformatics. *Briefings in bioinformatics* 14, 5 (2013), 633–647. doi:10.1093/bib/bbt020

[16] Edsger W Dijkstra. 2022. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*. 287–290. doi:10.1007/BF01386390

[17] Max Doblas, Oscar Lostes-Cazorla, Quim Aguado-Puig, Cristian Iñiguez, Miquel Moreto, and Santiago Marco-Sola. 2025. QuickEd: high-performance exact sequence alignment based on bound-and-align. *Bioinformatics* 41, 3 (2025), btaf112. doi:10.1093/bioinformatics/btaf112

[18] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press. doi:10.1017/CBO9780511790492

[19] Sean R. Eddy. 1998. Profile hidden Markov models. *Bioinformatics (Oxford, England)* 14, 9 (1998), 755–763. doi:10.1093/bioinformatics/14.9.755

[20] Sean R Eddy. 2011. Accelerated profile HMM searches. *PLoS computational biology* 7, 10 (2011), e1002195. doi:10.1371/journal.pcbi.1002195

[21] Jason Eisner and Nathaniel W Filardo. 2010. Dyna: Extending datalog for modern AI. In *International Datalog 2.0 Workshop*. Springer, 181–220. doi:10.1007/978-3-642-24206-9_11

[22] Michael Farrar. 2007. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* 23, 2 (2007), 156–161. doi:10.1093/bioinformatics/btl582

[23] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. doi:10.1007/BF01407931

[24] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming* 21, 5 (1992), 313–347. doi:10.1007/BF01407835

[25] Joseph Felsenstein. 1981. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of molecular evolution* 17, 6 (1981), 368–376. doi:10.1007/BF01734359

[26] Yan Gao, Yongzhuang Liu, Yanmei Ma, Bo Liu, Yadong Wang, and Yi Xing. 2021. abPOA: an SIMD-based C library for fast partial order alignment using adaptive band. *Bioinformatics* 37, 15 (2021), 2209–2211. doi:10.1093/bioinformatics/btaa963

[27] Sneha D Goenka, Yatish Turakhia, Benedict Paten, and Mark Horowitz. 2020. SegAlign: A scalable GPU-based whole genome aligner. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13. doi:10.1109/SC41405.2020.00043

[28] Osamu Gotoh. 1982. An improved algorithm for matching biological sequences. *Journal of molecular biology* 162, 3 (1982), 705–708. doi:10.1016/0022-2836(82)90398-9

[29] Ragnar Groot Koerkamp and Pesho Ivanov. 2024. Exact global alignment using A* with chaining seed heuristic and match pruning. *Bioinformatics* 40, 3 (2024), btae032. doi:10.1093/bioinformatics/btae032

[30] Dan Gusfield. 1997. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News* 28, 4 (1997), 41–60. doi:10.1145/270563.571472

[31] Robert S Harris. 2007. *Improved pairwise alignment of genomic DNA*. The Pennsylvania State University.

[32] Liang Huang, He Zhang, Dezhong Deng, Kai Zhao, Kaibo Liu, David A Hendrix, and David H Mathews. 2019. LinearFold: linear-time approximate RNA folding by 5'-to-3'dynamic programming and beam search. *Bioinformatics* 35, 14 (2019), i295–i304. doi:10.1093/bioinformatics/btz375

[33] Chirag Jain, Luis M Rodriguez-R, Adam M Phillippy, Konstantinos T Konstantinidis, and Srinivas Aluru. 2018. High throughput ANI analysis of 90K prokaryotic genomes reveals clear species boundaries. *Nature communications* 9, 1 (2018), 5114. doi:10.1038/s41467-018-07641-9

[34] Saurabh Kalikar, Chirag Jain, Md Vasimuddin, and Sanchit Misra. 2022. Accelerating minimap2 for long-read sequencing applications on modern CPUs. *Nature Computational Science* 2, 2 (2022), 78–83. doi:10.1038/s43588-022-00201-8

[35] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29. doi:10.1145/3133901

[36] Konstantina Koliogeorgi, Sotirios Xydis, and Dimitrios Soudris. 2021. FPGA Acceleration of Short Read Alignment. In *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*. 1–2.

[37] Leslie Lamport. 1974. The parallel execution of DO loops. *Commun. ACM* 17, 2 (1974), 83–93. doi:10.1145/360827.360844

[38] Ben Langmead and Steven L Salzberg. 2012. Fast gapped-read alignment with Bowtie 2. *Nature methods* 9, 4 (2012), 357–359. doi:10.1038/nmeth.1923

[39] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100. doi:10.1093/bioinformatics/bty191

[40] Heng Li and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform. *bioinformatics* 25, 14 (2009), 1754–1760. doi:10.1093/bioinformatics/btp324

[41] Heng Li and Richard Durbin. 2010. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics* 26, 5 (2010), 589–595. doi:10.1093/bioinformatics/btp698

[42] Xingyu Liao, Wufei Zhu, Juexiao Zhou, Haoyang Li, Xiaopeng Xu, Bin Zhang, and Xin Gao. 2023. Repetitive DNA sequence detection and its role in the human genome. *Communications biology* 6, 1 (2023), 954. doi:10.1038/s42003-023-05322-y

[43] Yongchao Liu, Douglas L Maskell, and Bertil Schmidt. 2009. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC research notes* 2, 1 (2009), 73. doi:10.1186/1756-0500-2-73

[44] Zhuren Liu, Shouzhe Zhang, and Hui Zhao. 2024. Survey of hardware acceleration of genomic analysis. In *2024 IEEE 17th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 540–547. doi:10.1109/MCSoC64144.2024.00093

[45] Dirk M Lorenz, Alice Jeng, and Michael W Deem. 2011. The emergence of modularity in biological systems. *Physics of life reviews* 8, 2 (2011), 129–160. doi:10.1016/j.plrev.2011.02.003

[46] Kanak Mahadik, Christopher Wright, Jinyi Zhang, Milind Kulkarni, Saurabh Bagchi, and Somali Chaterji. 2016. Sarvavid: a domain specific language for developing scalable computational genomics applications. In *Proceedings of the 2016 international conference on supercomputing*. 1–12.

[47] Santiago Marco-Sola, Jordan M Eizenga, Andrea Guarracino, Benedict Paten, Erik Garrison, and Miquel Moreto. 2023. Optimal gap-affine alignment in O (s) space. *Bioinformatics* 39, 2 (2023), btad074. doi:10.1093/bioinformatics/btad074

[48] Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. 2021. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics* 37, 4 (2021), 456–463. doi:10.1093/bioinformatics/btaa777

[49] David H Mathews, Walter N Moss, and Douglas H Turner. 2010. Folding and finding RNA secondary structure. *Cold Spring Harbor perspectives in biology* 2, 12 (2010), a003665. doi:10.1101/cshperspect.a003665

[50] John S McCaskill. 1990. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers: original research on biomolecules* 29, 6-7 (1990), 1105–1119. doi:10.1002/bip.360290621

[51] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103. doi:10.7717/peerj-cs.103

[52] David W Mount. 2001. *Bioinformatics: sequence and genome analysis*. Vol. 564. Cold spring harbor laboratory press Cold Spring Harbor, NY.

[53] André Müller, Bertil Schmidt, Richard Membarth, Roland Leißa, and Sebastian Hack. 2022. Anyseq/gpu: a novel approach for faster sequence alignment on gpus. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–11. doi:10.1145/3524059.3532376

[54] Gene Myers. 1999. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)* 46, 3 (1999), 395–415. doi:10.1145/316542.316550

[55] National Human Genome Research Institute. [n. d.]. Base Pair (Genetics Glossary). https://www.genome.gov/genetics-glossary/Base-Pair. Accessed 2026-02-20.

[56] Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* 48, 3 (1970), 443–453. doi:10.1016/0022-2836(70)90057-4

[57] Ruth Nussinov and Ann B Jacobson. 1980. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences* 77, 11 (1980), 6309–6313. doi:10.1073/pnas.77.11.6309

[58] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. 2001. An Eulerian path approach to DNA fragment assembly. *Proceedings of the national academy of sciences* 98, 17 (2001), 9748–9753. doi:10.1073/pnas.171285098

[59] Minh Pham, Yicheng Tu, and Xiaoyi Lv. 2023. Accelerating BWA-MEM read mapping on GPUs. In *Proceedings of the 37th international conference on supercomputing*. 155–166. doi:10.1145/3577193.3593703

[60] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530. doi:10.1145/2491956.2462176

[61] Knut Reinert, Temesgen Hailemariam Dadi, Marcel Ehrhardt, Hannes Hauswedell, Svenja Mehringer, René Rahn, Jongkyu Kim, Christopher Pockrandt, Jörg Winkler, Enrico Siragusa, et al. 2017. The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. *Journal of biotechnology* 261 (2017), 157–168. doi:10.1016/j.jbiotec.2017.07.017

[62] Isidore Rigoutsos and Aris Floratos. 1998. Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm. *Bioinformatics (Oxford, England)* 14, 1 (1998), 55–67. doi:10.1093/bioinformatics/14.1.55

[63] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. 2004. Reducing storage requirements for biological sequence comparison. *Bioinformatics* 20, 18 (2004), 3363–3369. doi:10.1093/bioinformatics/bth408

[64] Naruya Saitou and Masatoshi Nei. 1987. The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees. *Molecular Biology and Evolution* 4, 4 (July 1987), 406–425. doi:10.1093/oxfordjournals.molbev.a040454

[65] Georg Sauthoff, Mathias Möhl, Stefan Janssen, and Robert Giegerich. 2013. Bellman's GAP—a language and compiler for dynamic programming in sequence analysis. *Bioinformatics* 29, 5 (2013), 551–560. doi:10.1093/bioinformatics/btt022

[66] Amir Shaikhha, Dan Suciu, Maximilian Schleich, and Hung Ngo. 2024. Optimizing nested recursive queries. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27. doi:10.1145/3639271

[67] Ariya Shajii, Ibrahim Numanagić, Riyadh Baghdadi, Bonnie Berger, and Saman Amarasinghe. 2019. Seq: a high-performance language for bioinformatics. *Proceedings of the ACM on programming languages* 3, OOPSLA (2019), 1–29. doi:10.1145/3360551

[68] Temple F Smith, Michael S Waterman, et al. 1981. Identification of common molecular subsequences. *Journal of molecular biology* 147, 1 (1981), 195–197. doi:10.1016/0022-2836(81)90087-5

[69] Martin Šošić and Mile Šikić. 2017. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics* 33, 9 (2017), 1394–1395. doi:10.1093/bioinformatics/btw753

[70] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85. doi:10.1109/CGO.2017.7863730

[71] Shiv Sundram, Muhammad Usman Tariq, and Fredrik Kjolstad. 2024. Compiling Recurrences over Dense and Sparse Arrays. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 250–275. doi:10.1145/3649820

[72] Hajime Suzuki and Masahiro Kasahara. 2018. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC bioinformatics* 19, Suppl 1 (2018), 45. doi:10.1186/s12859-018-2014-8

[73] Muhammad Usman Tariq, Shiv Sundram, and Fredrik Kjolstad. 2025. REPTILE: Performant Tiling of Recurrences. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 670–696. doi:10.1145/3763074

[74] Martina Tarozzi, Anna Bartoletti-Stella, Daniele Dall'Olio, Tommaso Matteuzzi, Simone Baiardi, Piero Parchi, Gastone Castellani, and Sabina Capellari. 2022. Identification of recurrent genetic patterns from targeted sequencing panels with advanced data science: a case-study on sporadic and genetic neurodegenerative diseases. *BMC medical genomics* 15, 1 (2022), 26. doi:10.1186/s12920-022-01173-4

[75] Yatish Turakhia, Sneha D Goenka, Gill Bejerano, and WIlliam J Dally. 2019. Darwin-WGA: A co-processor provides increased sensitivity in whole genome alignments with high speedup. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 359–372. doi:10.1109/HPCA.2019.00050

[76] Esko Ukkonen. 1985. Algorithms for approximate string matching. *Information and control* 64, 1-3 (1985), 100–118. doi:10.1016/S0019-9958(85)80046-2

[77] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018). doi:10.48550/arXiv.1802.04730

[78] Md Vasimuddin, Sanchit Misra, Heng Li, and Srinivas Aluru. 2019. Efficient architecture-aware acceleration of BWA-MEM for multicore systems. In *2019 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 314–324. doi:10.1109/IPDPS.2019.00041

[79] Bala Vinaithirthan, Shiv Sundram, Sneha Goenka, and Fredrik Kjolstad. 2026. *Filtr: Compiling Bioinformatics Recurrences Artifact*. doi:10.5281/zenodo.21938559

[80] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. 2015. Hardware-acceleration of short-read alignment based on the burrows-wheeler transform. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2015), 1358–1372. doi:10.1109/TPDS.2015.2444376

[81] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing recursive queries with progam synthesis. In *Proceedings of the 2022 International Conference on Management of Data*. 79–93. doi:10.1145/3514221.3517827

[82] Linyan Xue, Xiaoke Zhang, Fei Xie, Shuang Liu, and Peng Lin. 2019. Frequent patterns algorithm of biological sequences based on pattern prefix-tree. *International Journal of Computers Communications & Control* 14, 4 (2019), 574–589. doi:10.15837/ijccc.2019.4.3607

[83] Katherine Yelick, Aydın Buluç, Muaaz Awan, Ariful Azad, Benjamin Brock, Rob Egan, Saliya Ekanayake, Marquita Ellis, Evangelos Georganas, Giulia Guidi, et al. 2020. The parallelism motifs of genomic data analysis. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* 378, 2166 (2020), 20190394. doi:10.1098/rsta.2019.0394

[84] Alberto Zeni, Giulia Guidi, Marquita Ellis, Nan Ding, Marco D Santambrogio, Steven Hofmeyr, Aydın Buluç, Leonid Oliker, and Katherine Yelick. 2020. Logan: High-performance gpu-based x-drop long-read alignment. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 462–471. doi:10.1109/IPDPS47924.2020.00055

[85] Daniel R Zerbino and Ewan Birney. 2008. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research* 18, 5 (2008), 821–829. doi:10.1101/gr.074492.107

[86] Haowen Zhang, Chirag Jain, and Srinivas Aluru. 2020. A comprehensive evaluation of long read error correction methods. *BMC genomics* 21, Suppl 6 (2020), 889. doi:10.1186/s12864-020-07227-0

[87] He Zhang, Liang Zhang, David H Mathews, and Liang Huang. 2020. LinearPartition: linear-time approximation of RNA folding partition function and base-pairing probabilities. *Bioinformatics* 36, Supplement_1 (2020), i258–i267. doi:10.1093/bioinformatics/btaa460

[88] Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller. 2000. A greedy algorithm for aligning DNA sequences. *Journal of Computational biology* 7, 1-2 (2000), 203–214. doi:10.1089/10665270050081478

[89] Michael Zuker. 2003. Mfold web server for nucleic acid folding and hybridization prediction. *Nucleic acids research* 31, 13 (2003), 3406–3415. doi:10.1093/nar/gkg595

[90] Michael Zuker and Patrick Stiegler. 1981. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic acids research* 9, 1 (1981), 133–148. doi:10.1093/nar/9.1.133

Received 2026-03-17; accepted 2026-08-06