Vectron: A Dynamic Programming Auto-Vectorization Framework

by

Sourena Naser Moghaddasi
B.Sc., Sharif University of Technology, 2008
M.Sc., K.N. University of Technology, 2012

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Sourena Naser Moghaddas, 2024
University of Victoria

All rights reserved. This Thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Vectron: A Dynamic Programming Auto-Vectorization Framework

by

Sourena Naser Moghaddasi
B.Sc., Sharif University of Technology, 2008
M.Sc., K.N. University of Technology, 2012

Supervisory Committee

Dr. Ibrahim Numanagić, Supervisor
(Department of Computer Science)

Dr. Sean Chester, Departmental Member
(Department of Computer Science)

Dr. Tao Lu, Outside Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Ibrahim Numanagić, Supervisor
(Department of Computer Science)

Dr. Sean Chester, Departmental Member
(Department of Computer Science)

Dr. Tao Lu, Outside Member
(Department of Electrical and Computer Engineering)

ABSTRACT

Dynamic programming (DP) is a fundamental algorithmic strategy that decomposes large problems into manageable subproblems. It is a cornerstone of many important computational methods in diverse fields, especially in the field of computational genomics, where it is used for sequence comparison. However, as the scale of the data keeps increasing, these algorithms are becoming a major computational bottleneck, and there is a need for strategies that can improve their performance. Here, we present Vectron, a novel auto-vectorization suite that targets array-based DP implementations written in Python and converts them to efficient vectorized counterparts that can efficiently process multiple problem instances in parallel. Leveraging Single Instruction Multiple Data (SIMD) capabilities in modern CPUs, along with Graphics Processing Units (GPUs), Vectron delivers significant speedups, ranging from 10% to more than 20×, over the conventional C++ implementations and manually vectorized and domain-specific state-of-the-art implementations, without necessitating large algorithm or code changes. Vectron's generality enables automatic vectorization of any array-based DP algorithm and, as a result, presents an attractive solution to optimization challenges inherent to DP algorithms.

Contents

Supervisory Committee ii

Abstract iii

Contents iv

List of Tables vi

List of Figures viii

Acknowledgements ix

Dedication x

1 Introduction 1

2 Background 4

3 Example 9

4 Implementation 14
- 4.1 Recurrence analysis . . . 15
- 4.2 Memory Operations in Vectron . . . 17
- 4.3 Initialization and aggregation . . . 18
- 4.4 Scheduling . . . 19
- 4.5 GPU support . . . 22

5 Experiments 23
- 5.1 Experiment Scenarios . . . 23
- 5.2 CPU (SIMD) Results . . . 24
- 5.3 GPU Results . . . 26

6 Conclusion **30**

Bibliography **31**

A Additional Information **36**

A.1 Artifact Appendix 36

A.1.1 Abstract 36

A.1.2 Artifact check-list (meta-information) 36

A.1.3 Description 38

A.1.4 Installation 39

A.1.5 Experiment workflow 40

A.1.6 Evaluation and expected result 40

A.1.7 Experiment customization 41

A.2 Sample Source Code Appendix 41

List of Tables

Table 4.1 Overview of the expressions that can be vectorized within a recurrence block. The first four transformations are straightforward. Store transformations are useful when dealing with multiple DP matrices at the same time. Finally, the comparator transformations handle either inline `if-else` expressions or comparator functions m that perform the more complex logic based on the input elements. Each such comparator function is also vectorized if possible (thus `cmp_vec` instead of `cmp`). 17

Table 5.1 Supported algorithms by the evaluated state-of-the-art methods for Smith-Waterman calculation. Other tools were not evaluated as their source code was unavailable. 25

Table 5.2 Comparison of SIMD-vectorized Vectron DP implementations and the equivalent baseline C++ and Codon versions on a small set of 4,096 sequence pairs (top), medium set of 262,144 sequence pairs (middle), and a large set of $4,194,304$ sequence pairs (bottom). Each sequence in a pair is of length 512. All times are shown in seconds (lower is better). The best times are highlighted in boldface. We also show Vectron's speed-up factor next to each other implementation (higher is better). 26

Table 5.3 Performance of the state-of-the-art algorithms. The best values are highlighted in boldface. The methods are grouped depending if they are optimized for single-instance or multi-instance DP. . 27

Table 5.4 Runtime comparison of Vectron against RV on small (4,096 sequence pairs) dataset. All numbers are reported in seconds. . . . 28

Table 5.5 GPU performance comparison between Vectron and C++ versions. Both Vectron and CUDA are slower than the baseline C++ version on small loads due to the cost of copying values between the GPU and the main memory. 29
Table 5.6 Performance comparison between state-of-the-art GPU implementations and Vectron's SIMD and GPU implementations of Smith-Waterman with integer scoring. 29

List of Figures

Figure 3.1 Sequential vs. 3D Vectorized DP Structures 13

Figure 4.1 A cache-friendly 3D matrix structure and the inverted loop strategy that utilizes double buffering. (a) The initial iteration compares x_i^0 with all elements y_j, resulting in a scoring matrix that is inserted into the 3D structure. (b) The second iteration compares x_i^1 with all y_j, generating another matrix that is appended to the 3D structure. (c) The third iteration follows the same pattern as the previous iterations. However, in this case, the new matrix replaces the one from the first iteration in a rolling fashion. This mimics a double-buffering scheme, where the outdated matrix is discarded as it will not be needed later on. This technique helps maintain a minimal memory footprint while maximizing cache efficiency. 22

Figure 5.1 27

ACKNOWLEDGEMENTS

This master's thesis would not have been possible without the support and contributions of many individuals.

I am particularly indebted to Dr. Ibrahim Numanagić, whose expertise and consistent guidance were instrumental throughout the research process.

A special thank you goes to my wife, whose unwavering love and encouragement provided me with the strength to persevere.

I also want to acknowledge the faculty members of the Computer Science Department, whose knowledge and teachings greatly enriched my academic experience, as well as my colleagues in the lab for their valuable assistance.

Lastly, I am grateful to the University of Victoria for offering the necessary resources and facilities that greatly aided this research.

To everyone who contributed in various ways, your support has been deeply appreciated. Thank you for being part of this journey.

DEDICATION

To the pioneers in fields ranging from space exploration to biotechnology, this work is dedicated to you. Whether you're charting new territories among the stars, unravelling the complexities of our immune system, or optimizing logistics and supply chains, your relentless pursuit of excellence and innovation is inspiring. This thesis, which enhances the speed of solving Dynamic Programming problems, is a tribute to your work. May it empower your endeavours and contribute to breakthroughs across industries, paving the way for a brighter and more efficient future.

Chapter 1

Introduction

Dynamic programming (DP) is one of the fundamental strategies in algorithmic problem-solving that works by recursively breaking down complex problems into smaller subproblems that can be easily solved. It is widely used in many scientific fields, from economics Kruskal [1983] to bioinformatics Aluru [2005].

DP implementations most commonly use matrices or multi-dimensional arrays to store subproblem solutions. These matrices track values derived from the previous calculations and are typically populated through a set of nested loops, wherein the current element in each iteration is calculated as a function of the previously computed elements in the matrix. The execution is typically sequential and cannot be easily parallelized due to interlocking data dependencies (Okada et al. [2015]).

To address this bottleneck and improve the performance of DP algorithms, various problem-specific strategies have emerged. Some of them target a particular DP recurrence and prune unnecessary calculations by precomputing key values in advance (Galil and Park [1989], Yao [1980]). Others explore parallel processing and vectorization—typically through data-level parallelism—to leverage hardware advancements for speed. This includes utilizing Single Instruction Multiple Data (SIMD) capabilities in modern CPUs, Field-Programmable Gate Arrays (FPGAs), or Graphics Processing Units (GPUs).

SIMD approaches offer the most limited capability for bundling repetitive calculations but are widely supported by modern CPUs and require no specialized hardware (Rudnicki et al. [2009]), FPGAs offer customizable hardware solutions tailored for specific algorithms Settle [2013], while GPUs excel at parallel processing for floating-point values, handling multiple tasks simultaneously Stoyanova et al. [2014]. Note that the latter strategy would require specialized hardware (a GPU), and FPGAs

necessitate custom and oftentimes expensive hardware. Yet, all of these strategies are used to solve a single instance of a specific DP problem as efficiently as possible and are, as such, fundamentally bounded by the sequential data bottlenecks inherent to the problem itself. They are also often hard to implement and maintain, especially if targeting multiple platforms is desired, and often necessitate significant changes to the underlying recurrence or data layout to expose the data-level parallelism, which can be problematic due to increased complexity and potential performance trade-offs across different architectures.

In many cases, however, the end user is not interested in solving a single instance of a DP problem as efficiently as possible but instead in solving many instances of a single DP problem in parallel. The quintessential example is the "foundational" bioinformatics workflow for aligning short DNA sequences (strings) to the reference genome. Here, billions of short sequences are matched to the reference sequence via a dynamic programming strategy known as Smith-Waterman alignment (Smith and Waterman [1981]) in no particular order. Such problems are, in theory, *embarrassingly parallelizable* since they have no inter-dependencies between multiple problem instances (Rognes [2011]). As these workflows take hours or days on modern CPU machines to complete (Li and Durbin [2009]), any performance improvements to the alignment procedure are more than welcome.

Here, we focus on strategies that exploit data-level parallelism through SIMD (e.g., SSE or AVX) or SIMT (e.g., CUDA) intrinsic, which can significantly improve the performance of solving a large number of instances of a given DP problem. More specifically, here we introduce Vectron, a compiler analysis and auto-vectorization suite for high-level, Python-like code that automatically analyzes an array-based DP formulation and transforms it to a high-performance variant that maximizes the amount of data-level parallelism when done over multiple instances. This strategy has already been applied in various domain-specific applications Miller and Myers [1988], Sankoff [1972], Awan et al. [2020], Shajii et al. [2021], albeit in a manual and hard-to-generalize fashion for a specific subset of DP problems.

Vectron is based on Codon (Shajii et al. [2023]), an ahead-of-time compilation framework for Python-like code that provides the simplicity and familiarity of Python's syntax and, at the same time, enables implementation of complex compiler analysis and transformation passes on top of Python code. We used Vectron to implement various dynamic programming schemes and compared them against other state-of-the-art, manually optimized solutions that are optimized for either a single or

multiple instances of a DP problem. On average, Vectron achieved a speedup of up to 17.73× when compared to other solutions without requiring end-users to implement any manual vectorization schemes or perform large-scale refactoring.
In short, we provide the following contributions:

1. A compiler pass for automatically exposing data-level parallelism (auto-vectorization) in dynamic programming procedures across multiple instances;
2. Auto-vectorization for both CPUs (SIMD) and GPUs (SIMT) targets; and
3. A performance benchmark of the state-of-the-art implementations of DP algorithms (Smith-Waterman, Needleman-Wunsch, Longest Common Subsequence, Manhattan Tourist, Hamming Distance and others) and their counterparts in Python (Vectron) and plain C++.

Vectron is open-source and available as a Codon plugin on GitHub 0xTCG [2024].

Chapter 2

Background

While the scope of the thesis is general, most of the past work in improving the performance of DP schemes stems from the field of computational genomics due to sequence alignment being both a cornerstone method and the largest bottleneck when doing any sort of genome analysis Smith and Waterman [1981], Needleman and Wunsch [1970]. Thus, this section will mostly deal with sequence alignment-based schemes.

In the context of sequence alignment, dynamic programming provides a systematic way to find the optimal alignment between two sequences (strings of characters) by constructing a matrix that quantifies the similarity or difference between them. Each cell in the matrix represents a specific comparison of characters, and the value stored in that cell reflects the cost (or score) associated with aligning those characters, considering potential gaps (insertions or deletions).

Sequence alignment algorithms are designed to identify the best possible alignment between biological sequences, such as DNA, RNA, or protein sequences. The most widely used algorithms are:

- Needleman-Wunsch Algorithm: This algorithm computes the optimal global alignment of two sequences, ensuring that every character in both sequences is aligned. It does so by filling out a matrix based on a scoring scheme that accounts for matches, mismatches, and gap penalties.

- Smith-Waterman Algorithm: In contrast, this algorithm focuses on local alignment, identifying the most similar subsequences between the two sequences. It is particularly useful when sequences may have regions of high similarity within larger sequences that differ substantially.

Both algorithms rely on recursive relationships to fill their respective matrices, which makes them computationally intensive, particularly for long sequences. This quadratic time complexity ($O(n \times m)$, where n and m are the lengths of the sequences) can lead to significant performance challenges, prompting the need for parallelization and optimization strategies, such as those discussed in this thesis.

The Needleman-Wunsch DP recurrence forms the basis of all later alignment methods and was developed by Needleman and Wunsch [1970]. Sankoff [1972] addressed alignment complexity by considering constraints like deletions and insertions, while Waterman et al. [1976] established mathematical foundations and metrics for sequence comparison. Smith and Waterman [1981] proposed the Smith-Waterman local alignment algorithm, an extension of the Needleman-Wunsch algorithm that can identify similar regions within two sequences. Gotoh [1982] further refined Smith-Waterman by introducing the concept of affine gap penalties. Other significant developments include the expanded theoretical foundations of sequence alignment (Kruskal [1983]), improved statistical models for sequence alignment (Waterman [1984]), more complex alignment models that incorporate concave gap costs and enable tailored alignment scoring schemes (Miller and Myers [1988]), and finally, the introduction of generalized affine gap costs to refine protein similarity measurements (Altschul [1998]). These approaches follow the same pattern established by the original Needleman-Wunsch strategy and differ mostly in the matrix setup and the details of the underlying DP recurrence.

The proposed methods have been proven to have at least a quadratic time complexity Backurs and Indyk [2015]. This limitation made the alignment methods a main bottleneck of genomics pipelines and thus led to the exploration of various strategies for parallel computation of Needleman-Wunsch or Smith-Waterman scores. Early studies emphasized the emerging role of data parallelism in speeding up protein/DNA sequence alignment. Alpern et al. [1995], for instance, paved the way for modern SIMD and GPU-based implementations, while Knee [1997] provided a parallel computation model for sequence alignment. Myers [1999] began with a novel approach for approximate string matching through DP that leverages fast bit-vector operations to accelerate string matching, while modified methods such as Striped Smith-Waterman greatly improved database search efficiency Farrar [2007]. Since then, numerous libraries and methods have focused on improving the alignment performance of a single pair sequence alignment via SIMD. Examples include SeqAn (Döring et al. [2008]), a generic C++ library for sequence analysis; SSW (Zhao et al. [2013]), a SIMD Smith-

Waterman C/C++ library; and Edlib (Šošić and Šikić [2017]), a C/C++ library for fast, exact sequence alignment that uses SIMD internally for improved performance. Suzuki and Kasahara [2018] presented a novel Smith-Waterman variant that uses difference recurrence relations for faster semi-global alignment of long sequences, thus contributing to the efficiency of sequence alignment. This was implemented in one of the currently fastest alignment libraries, KSW2 (Li [2020]), used by the minimap2 sequence aligner (Li [2018]). Aalign (Hou et al. [2016]) presented a technique capable of performing the Smith-Waterman and Needleman-Wunsch algorithms, achieving more than 10× speedups over sequential versions on Intel Haswell CPUs. Parasail (Daily [2016]) is an implementation capable of performing various local, semi-global, and global Smith-Waterman and Needleman-Wunsch algorithms in a SIMD manner and has a reported speed of around 6 GCUPS for sequence pairs of length 512. Finally, Jararweh et al. [2019] proposed a Needleman-Wunsch algorithm that can achieve nearly 15× speedup over the sequential version on Intel Core processors.

In addition to advancements targeting single query-target alignments, a few methods also focused on optimizing many-targets versus many-query alignment in parallel. This was originally considered in Alpern et al. [1995] in the context of microparallelism and high-performance protein matching. Later on, SWIPE (Rognes [2011]) introduced the concept of *sequence inter-alignment*—calculating multiple Smith-Waterman alignments independently in parallel—and proposed a SIMD-based solution to improve its performance. Another similar implementation is the many-to-many version of SeqAn (Rahn et al. [2018]) that accelerates sequence alignment in original SeqAn (Döring et al. [2008]) across many instances through vectorization and multi-threading. Finally, Shajii et al. [2019] introduced Seq, a domain-specific language designed specifically for bioinformatics applications, and within it expanded the inter-alignment SIMD strategy from Rognes [2011] to various variants of sequence alignment problems. Other approaches, such as Yin et al. [2019], showcased new possibilities that combine high-performance SIMD sequence alignment with massively parallel execution strategies in clustered computing environments.

Specialized Smith-Waterman strategies, as reviewed by Barnes [2020], have also moved into GPU acceleration strategies. Liao et al. [2018] introduced a tailored Smith-Waterman algorithm for longer reads, addressing efficiency issues with long sequences. Okada et al. [2015] presented SW# that accelerates the Smith-Waterman algorithm through interpair pruning and band optimization, enabling efficient large-scale sequence alignment by reducing computational complexity and memory usage.

ADEPT (Awan et al. [2020]) is another innovation in GPU-accelerated sequence alignment strategies that leverages GPU-specific optimizations and scalable driver mechanisms to achieve remarkable performance boosts of up to 360 GCUPS for protein-based and DNA-based datasets on a supercomputer.

Polyhedral optimization is another technique that has been explored to improve the performance of dynamic programming algorithms. It involves representing loops and their dependencies using mathematical structures called polyhedra, which allow for sophisticated transformations such as loop tiling, fusion, and parallelization. An example of polyhedral optimization is the Pluto compiler, which automatically parallelizes and optimizes program loops for better performance (Bondhugula et al. [2008]). This technique can be integrated with compilers like LLVM to enhance the efficiency of generated code.

Some other studies have focused on optimizing other DP algorithms on sequences. For instance, Kim et al. [2018] explore DP-based techniques tailored for high-throughput DNA sequencing, achieving a 50% reduction in alignment time for large sequencing datasets. Yin et al. [2019] presents XLCS, a novel bit-parallel algorithm tailored for Xeon Phi clusters, optimizing the Longest Common Subsequence (LCS) problem. The approach leverages bitwise operations and parallel processing, demonstrating $> 3\times$ speedups versus one-versus-many query-targets equivalents. Liu et al. [2019] propose a scalable approach using divide-and-conquer DP, enabling alignment of large datasets with reduced memory requirements and improved scalability. Zhang et al. [2020] present an efficient algorithm for many-to-many alignment (a DP method distinct from traditional sequence alignment) using progressive DP, demonstrating a 30% reduction in computation time compared to state-of-the-art methods, and authors of Wang et al. [2021] introduce parallelizable and memory-efficient DP algorithms for multiple sequence alignment, achieving up to a twofold improvement in execution time and memory usage compared to existing approaches.

Most of the methods mentioned earlier are implemented as a set of low-level SIMD or GPU intrinsics manually optimized for a particular architecture (typically Intel's SSE or AVX). Auto-vectorization schemes, on the other hand, are able to maintain code simplicity and target various architectures by leveraging compiler optimizations to generate SIMD/SIMT instructions automatically. They are widely used in many domains and by many popular compiler frameworks, such as LLVM (Lattner and Adve [2002]) or extensions like RV (Schryver et al. [2024]). These approaches are able to automatically vectorize many kinds of loops and procedures without explicit user

intervention. However, neither of these approaches is able to vectorize complex loop structures common in dynamic programming methods.

Chapter 3

Example

We will use the Needleman-Wunsch dynamic programming algorithm that computes the similarity between two strings as a lead example to motivate and explain the inner workings of Vectron. Needleman-Wunsch is a string matching method that computes the minimum number of character substitutions, insertions, and deletions needed to convert string T into Q—in other words, their edit distance. It and its close cousins, such as Smith-Waterman and gapped local alignment, are ubiquitous in bioinformatics Smith and Waterman [1981] and are also major bottlenecks in processing high-throughput DNA sequencing data due to their quadratic complexity Backurs and Indyk [2015]. This algorithm follows the generic DP structure shown in Algorithm 1, in which subproblems are evaluated in a bottom-up fashion, and arrays (usually matrices) are used to store subproblem solutions.

Algorithm 1 A generic dynamic programming (DP) problem. It uses matrices to track the optimal score between subproblems j of the input problem i. In the case of string algorithms, subproblems are typically substrings of the input string. The matrix dimensions typically depend on the size of the input problem (e.g., string length). The problem itself is defined as a recurrence that utilizes previously computed subproblem solutions—stored in the DP matrix—to compute a solution to the larger problem. The matrix is typically populated by iterating through a list of subproblems in a bottom-up fashion.

```
1: Initialize M                        ▷ an n-dimensional array that tracks subproblems
2: for all subproblem i do
3:     M(i) = f(M(j) | j ⊂ i)          ▷ j is a subproblem of i that has already been computed
4: end for
5: return M(final)
```

Given two strings $T = t_1 \dots t_m$ and $Q = q_1 \dots q_n$, the Needleman-Wunsch algorithm computes a matrix M in a bottom-up manner as follows. For each prefix $T[1 \dots i]$ and $Q[1 \dots j]$ of T and Q, respectively, Needleman-Wunsch computes $M_{i,j}$—the smallest edit distance between these two prefixes—by utilizing previously computed results through the following recurrence:

$$M_{i,j} = \min \{ M_{i-1,j} + \gamma, M_{i,j-1} + \gamma, M_{i-1,j-1} + \delta(t_i, q_j) \} .$$

Here, γ is a gap penalty for introducing character insertion or deletion, while $\delta(a, b)$ computes the substitution penalty between the characters a and b. Typically, $\gamma = 1$ and $\delta(a, b) = 1$ if $a \neq b$ or 0 otherwise. If either i or j are zero, $M_{i,j} = \gamma \min\{i, j\}$.

Code Block 1 The Needleman Wunsch Algorithm and its components. The *initialization block* (lines 1–2; yellow box) initializes the DP matrix M. The *loop block* (lines 3–9; green box) iterates through the subproblems. The *recurrence block* within it (lines 5–9; blue box) populates M via the DP recurrence, while the *aggregation block* (line 10; red box) returns the desired score.

```
1  M = [[(i * gap if j == 0 else j * gap if i == 0 else 0) for j in
2  range(len(q) + 1)] for i in range(len(t) + 1)]
3  for i in range(1, len(t) + 1):
4    for j in range(1, len(q) + 1):
5      M[i][j] = min(
6        M[i][j-1] + gap,
7        M[i-1][j] + gap,
8        M[i-1][j-1] +
9          (mis if t[i-1] != q[j-1] else 0))
10 return M[-1][-1]
```

A succinct Python implementation of Needleman-Wunsch is shown in Code Block 1. Here, the *kernel* procedure (referred to as $\mathcal{K}$) computes a score for a single instance of a DP problem (i.e., a pair of sequences) and can be partitioned into four blocks. The *initialization block* initializes the matrix M (in this case, the scores on its boundary). The two loops iterate through the subproblems, while the innermost recurrence populates the matrix M via simple expression within the *reccurence block*. Finally, the desired value is returned in the last line, which is considered the *aggregation block*.

This algorithm comes in many other flavors. For example, genomics tools often use the Smith-Waterman algorithm with Gotoh scoring Gotoh [1982] to compute the local edit distance with affine gap penalty between strings T and Q. This approach

requires the computation of three matrices: E, F, and H via the following recurrences:

$$\begin{aligned} E_{i,j} &= \max\left\{E_{i,j-1} + \gamma_E, H_{i,j-1} + \gamma_O + \gamma_E\right\}; \\ F_{i,j} &= \max\left\{F_{i-1,j} + \gamma_E, H_{i-1,j} + \gamma_O + \gamma_E\right\}; \\ H_{i,j} &= \max\left\{E_{i,j}, F_{i,j}, H_{i-1,j-1} + \delta(q_i, t_j), 0\right\}. \end{aligned}$$

Here, matrices E and F track the lengths of gaps—i.e., insertions and deletions—in T and Q and ensure that starting a gap is penalized differently (with γ_O) than extending a gap that has already been initiated (with γ_E). H, on the other hand, tracks substitutions and gaps and picks the best choice at each step. Note that the basic structure—initialization, iteration and recurrence—still remains the same despite the increase in complexity of each individual block. An example Python implementation of this method is shown in Appendix 1.

Here, we are not interested in calling the kernel once but many times on a large set of string pairs. More formally, we are interested in computing kernels $\mathcal{K}(T_k, Q_k)$ for each pair (T_k, Q_k) where $T_k \in \mathcal{T} = T_1, \ldots, T_p$ and $Q_k \in \mathcal{Q} = Q_1, \ldots, Q_p$. This can be naïvely done by iterating over all pairs (T_k, Q_k) and calling the kernel $\mathcal{K}(T_k, Q_k)$ separately (Code Block 2). However, this is often not efficient if the number of pairs is large (in bioinformatics, this number is often measured in billions).

Code Block 2 Naïve implementation of Needleman-Wunsch on many sequence pairs.

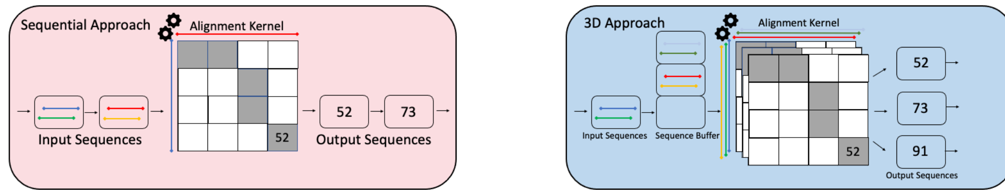
```
1 def process_naive(T, Q):
2   for t, q in zip(T, Q):
3     nw_kernel(t, q)
```

Instead, here we want to improve the performance by relying on *inter-sequence alignment* Rognes [2011]. More specifically, we want to auto-vectorize the loop in Code Block 1 so that a whole set of pairs can be processed in a single-instruction manner (such as SIMD). An illustration of this strategy is shown in Figure 1, while a simple transformation is shown in Code Block 3. Note that the basic structure of the kernel `nw_kernel` remains the same; the only things that change are the fact that each value is replaced with a vector of values and that each operation deals with vectors instead of single values. The vector operations are typically done in SIMD or SIMT fashion through the underlying `Vec` type.

Code Block 3 A high-level overview of the vectorized kernel for the original Needleman-Wunsch kernel (`nw_kernel` in Code Block 1). This kernel computes the alignment scores for many sequence pairs in parallel. It operates on vectors instead of single values, and all recurrence operations in lines 20–29 are vectorized operations. Vectors are represented as Codon's `Vec` type that has overloaded operators for each arithmetic operation (e.g., addition, minimization etc.).

```
1 def nw_simd_kernel(T, Q, gap=1, mis=1):
2   # We assume that sequence lengths are identical
3   l = len(T[0])  # Sequence length
4   np = len(T)    # Number of pairs
5   w = 8          # Vector width
6
7   # 3D matrix (a matrix for each sequence pair)
8   M = [
9     [[gap * min(i, j)] * np for i in range(l + 1)]
10    for j in range(l + 1)
11  ]
12  for k in range(0, np, w):
13    # M[i][j][s] returns a vector spanning elements
14    # in slice 's'
15    for i in range(1, l + 1):
16      s = slice(k + i, k + i + w)
17      for j in range(1, l + 1):
18        M[i][j][s] = min(
19          # Constants are vectorized via 'Vec(...)'
20          M[i][j-1][s] + Vec(gap, w),
21          M[i-1][j][s] + Vec(gap, w),
22          M[i-1][j-1][s] + Vec.cmp(
23            T[s][i-1], Q[s][j-1],
24            # 0 if above are equal, 'mis' otherwise
25            0, mis
26          )
27        )
28  return M[-1][-1] # final scores for all pairs
```

Figure 3.1: Sequential vs. 3D Vectorized DP Structures



(a) The sequential structure of DP algorithms where each pair is aligned sequentially.

(b) The Vectorized 3D Structure of SIMD DP Implementations where pairs of sequences are aligned together simultaneously.

Despite the conceptual simplicity of this approach, the proper implementation is not straightforward. Thus, in the upcoming section, we will describe Vectron, a compiler transformation pass that takes in Code Block 1 and produces its highly efficient vectorized equivalent akin to Code Block 3.

Chapter 4

Implementation

Vectron is implemented as a Codon (Shajii et al. [2023]) plugin and, as such, includes a library and a set of compiler passes for Python code analysis. Codon is an ahead-of-time compiler for Python-like code built on top of the LLVM compiler framework (Lattner and Adve [2002]). In addition to being strongly typed, having low overhead, and using a highly performant LLVM backend that enables various low-level code optimizations (such as inlining, efficient arithmetics, coroutines, lazy execution, automatic simple loop vectorization and so on), Codon also provides Codon IR, a specialized intermediate representation (IR) that allows easy implementation of code analysis and optimization passes that can operate on top of high-level Pythonic constructs.

Vectron starts by analyzing all functions decorated with the `@vectron` decorator that contains a single instance DP kernel. For each such kernel, Vectron generates a new procedure that takes a list of instances as input and performs a set of specific operations in parallel on these instances.

Kernel analysis begins by dividing the kernel function into four major blocks (see Code Block 1 for an overview). The first block—*initialization block*—deals with the creation and initialization of a DP matrix. Vectron considers anything before the main set of `for` loops as a part of the initialization block. The second block—*loop block*—is the core part of a DP method responsible for populating the DP matrix. It consists of a series of nested `for` loops and a DP recurrence. DP recurrence is considered as a *recurrence block*. Vectron assumes that everything within the innermost for loop is a recurrence block. Finally, the *aggregation block* aggregates and returns the final result from the calculated matrix. Vectron treats anything after the iteration block as the aggregation block.

4.1 Recurrence analysis

The main Vectron analysis pass consists of identifying and analyzing the iteration and recurrence blocks within a DP kernel. The pass begins by detecting a block of the nested for loops within the function. Once found, Vectron identifies the iteration behavior by extracting `start`, `step`, and `stop` bounds of each loop. Typically, `stop` is a constant or the length of an instance (e.g., string) plus or minus a constant, while the `start` and `step` values are constants. However, Vectron supports more complex custom ranges within the innermost loop. One example of this is the banded Smith-Waterman algorithm that only populates M within a diagonal band of a certain, user-defined length.

Once the loop range values have been identified, the pass moves on to the recurrence within the innermost `for` loop. There, it analyzes the recurrence expression and replaces it with the vectorized equivalent; in the case of SIMD, Vectron uses the `Vec` type and the associated SIMD intrinsics from Codon's SIMD library.

A recurrence expression is any expression that minimizes or maximizes something via `min` or `max` operator. This expression is itself composed of various subexpressions that are typically arguments to the `min`/`max` operators. Vectron vectorizes these subexpressions that are either (1) constants, (2) arithmetic operations on top of a previously calculated DP matrix value, (3) `min`/`max` operators, (4) store (assignment) operator, (5) comparator function, or (4) an arithmetic combination of previous expressions. The exact forms of these expressions and the associated transformations are provided in Table 1.

Upon analyzing the loop block, Vectron instantiates a separate loop block that iterates over a set of instances and auto-vectorizes the recurrence to perform each step in a data-parallel manner through either SIMD or SIMT (GPU) strategy. A user can also choose the exact SIMD target (e.g., SSE 4, AVX-512 etc.). An example of the completely transformed kernel for the Needleman-Wunsch algorithm is illustrated in Code Block 4.

Code Block 4 A simplified version of Vectron's SIMDified recurrence block for Needleman-Wunsch kernel in Code Block 1. The loop variables (e.g., `start1`, `stop1`, `step1`) as well as offsets (e.g., -1 or 0 in `M[i-1][j+0]`) are extracted from the userś code. Each operation operates on a vector of size `w` that begins at index `v`. `Vec.get(l, st, len)` initializes a SIMD vector from a list `l[st:st+len]`, while `Vec.get(const, len)` makes a SIMD vector of size `len` filled with `const`. Here, Vectron uses pre-processed array pointers `Mp`, `Tp` and `Qp` instead of `M`, `T` and `Q`, respectively. Note that we omitted some implementation details due to space constraints; for example, Vectron does not call `Vec.get`—an operation that copies `w` values to a SIMD register when called—in every loop pass, but instead shifts the corresponding vector and only appends new values in a streaming fashion. Also, note that the `Mp`'s indices are swapped when compared to the original `M` for improved cache locality. Other vector operations are outlined in Table 4.1.

```
1  start1, stop1, step1 = 1, len(t)+1, 1
2  start2, stop2, step2 = 1, len(q)+1, 1
3  for i in range(start1, stop1, step1):
4    for j in range(start2, stop2, step2):
5      for v in range(0, (len(t) + w - 1) // w * w, w):
6        Vec.store(
7          Mp[j][i], v, w,  # store to Mp[j][i][v:v+w]
8          Vec.__max__(
9            Vec.get(Mp[j-1][i-1], v, w) +
10             Vec.cmp(
11               Vec.get(Tp[i-1], v, w),
12               Vec.get(Qp[j-1], v, w),
13               0, 1),
14           Vec.__max__(
15             Vec.__add__(
16               Vec.get(Mp[j-1][i], v, w),
17               Vec.get(gap, w)),
18             Vec.__add__(
19               Vec.get(Mp[j][i-1], v, w),
20               Vec.get(gap, w)))))
```

Expression e	Sequential Form	Vectorized Form
Simple expressions:		
c (constant)	c	Vec(c, w)
$M_{i,j}$ (DP matrix element)	M[i][j]	Vec(M[i][j], k, k+w)
$M_{i,j} \circ c$ ($\circ \in \{+,-,*,/\}$)	M[i][j] op x	Vec.__op__(Vec(M[i][j], k, k+w), Vec(c)) (op∈ {add, sub, mul, div})
$\min\{e_1, e_2\}$	min(e1, e2)	Vec.__min__(e1, e2)
$\max\{e_1, e_2\}$	max(e1, e2)	Vec.__max__(e1, e2)
Store expressions:		
$H_{i,j} = e$	(H[i][j] := e)	Vec.store(H[i][j], k, k+w, e)
Comparators: compare target element t_i and query element q_j, return score based on comparison.		
$\mathbf{1}(t_i, q_j)$	1 if t[i] == q[j] else 0	Vec.cmp(Vec(T, k, k+w),Vec(Q, k, k+w),1, 0)
$m(i, j)$	cmp(t[i], q[j], args)	cmp_vec(Vec(T, k, k+w),Vec(Q, k, k+w),args)

Table 4.1: Overview of the expressions that can be vectorized within a recurrence block. The first four transformations are straightforward. Store transformations are useful when dealing with multiple DP matrices at the same time. Finally, the comparator transformations handle either inline `if-else` expressions or comparator functions m that perform the more complex logic based on the input elements. Each such comparator function is also vectorized if possible (thus `cmp_vec` instead of `cmp`).

4.2 Memory Operations in Vectron

In Vectron, efficient memory operations are crucial for optimizing performance, particularly when dealing with large data sets in dynamic programming algorithms. This section outlines how Vectron handles loading values from memory and setting values into registers, ensuring minimal overhead and maximum throughput.

Vectron's CPU mode employs SIMD (Single Instruction, Multiple Data) instructions to facilitate these memory operations, allowing it to load and set multiple data elements simultaneously. For instance, a sample unaligned load operation is illustrated LLVM Code Block 1.

LLVM Code Block 1 LLVM implementation of the unaligned loading function `_mm512_loadu_epi16`, which retrieves 32 packed 16-bit integers from an unaligned memory address into a 512-bit vector register.

```
1 @llvm
2 def _mm512_loadu_epi16(data: Ptr[i16]) -> Vec[i16, 32]:
3     %0 = bitcast i16* %data to <32 x i16>*
4     %1 = load <32 x i16>, <32 x i16>* %0
5     ret <32 x i16> %1
```

The `_mm512_loadu_epi16` function loads 32 packed 16-bit integers from an unaligned memory address into a 512-bit vector register. This method is particularly useful in scenarios where data may not be aligned to the typical memory boundaries, reducing the need for additional alignment operations.

In addition to loading values, Vectron also provides efficient mechanisms for setting data within registers. The following function demonstrates how Vectron can set all elements of a vector register to a specific value:

LLVM Code Block 2 LLVM implementation of the _mm512_set1_epi16 function, which sets all elements of a 512-bit vector to a specified 16-bit integer value. This operation initializes the vector with a consistent value, optimizing subsequent computations in Vectron.

```
1 @llvm
2 def _mm512_set1_epi16(val: i16) -> Vec[i16, 32]:
3     %0 = insertelement <32 x i16> undef, i16 %val, i16 0
4     %1 = shufflevector <32 x i16> %0, <32 x i16> undef,
5     <32 x i32> zeroinitializer
6     ret <32 x i16> %1
```

The _mm512_set1_epi16 function takes a single 16-bit integer value and sets all 32 packed 16-bit integers in the vector register to this value. This operation is crucial for initializing registers with a consistent state and optimizing the setup for subsequent computations.

Vectron efficiently manages memory interactions by leveraging these unaligned loading and setting operations along with similar storing commands, thereby enhancing performance for dynamic programming tasks, specifically for scheduling, initialization and aggregation operations.

4.3 Initialization and aggregation

Vectron treats any code before the main loops as the initialization block where the DP matrices are built. The user can build many different matrices in this block through conventional methods. Vectron will attempt to vectorize the initialization list comprehensions if possible; if not, it will build matrices as-is. Following the initialization, Vectron will identify the DP matrices and translate them to a semi-3D cache-friendly vectorized structure that will be used by the vectorized kernel. This structure will be explained in the next section.

The aggregation block typically contains a return statement that can be easily vectorized. In some instances, this block can contain a more complex set of statements or call a special *aggregation* function (decorated with `vectron_aggre- gate`), which performs additional operations on top of the final matrix score (e.g., z-drop score

processing implemented by many Smith-Waterman methods Li [2020]). Aggregation functions are also vectorized whenever possible.

Code Blocks 5 and 6 illustrate the SIMDified versions of initialization and aggregation blocks in the Needleman-Wunsch kernel.

Code Block 5 The SIMDified version of the initialization block for the Needleman Wunsch kernel from Code Block 1. Here Vectron targets AVX2 256-bit registers and fills each such register with sixteen 16-bit integers. Lines 2–6 initialize the boundaries of a DP matrix. Lines 7–9 initialize a matrix `Mp` that keeps SIMD vectors. Note that Vectron does not create a complete 3D matrix structure due to memory constraints but only keeps the last n queries (where n is the `band` variable in code and is set to 1 in case of Needleman-Wunsch) in the 3D structure. This also improves the cache performance. The last n queries are then iteratively rolled over in `Mp` in the outermost iteration within the SIMDified loop block (Code Block 4).

```
1 w = 16  # vector width
2 M_init = [[0 if (i == 0 and j == 0) else (i * gap)
3             if (j == 0 and i > 0)  else (j * gap)
4             if (i == 0 and j > 0)  else 0
5            for j in range(len(q) + 1)]
6           for i in range(len(q) + 1)]
7 Mp = [[Vec(M_init[i][k], j, w, dtype=i16)
8        for j in range(0, len(q) + 1, w)]
9       for k in range(band)]
```

Code Block 6 The transformed version of the aggregation block for the Needleman-Wunsch kernel from Code Block 1. This block extracts the final values from the vectorized DP matrix `Mp` and returns them as a list. Note that this block can contain more advanced operations; in that case, Vectron will attempt to vectorize all such operations.

```
1 scores = [0 for i in np]  # np: number of pairs
2 for i in range(0, np, w):
3   scores[i:i+w] = Vec.to_list(Mp[-1][-1], i)
4 return scores
```

4.4 Scheduling

Finally, Vectron integrates previous steps into a scheduler function that processes sequence pairs and dispatches them to the vectorized kernel. Currently, Vectron's

scheduler groups together pairs into blocks that are all to be executed in a data-parallel fashion. Typically, the scheduler groups n pairs into n/v blocks, where v is the maximum parallel throughput (e.g., v is 16 for 256-bit AVX2 vectors that contain packed 16-bit integer values). Each block will form a 3D structure that will be handled by the vectorized kernel. The SIMD version of the scheduler for the Needleman-Wunsch algorithm is depicted in Code Block 7.

Code Block 7 A Vectron scheduler for Needleman-Wunsch kernel under AVX2 SIMD setting. In this block, the `t_vec` and `q_vec` vectors are initialized as SIMD vectors, and the SIMDified kernel `simd_kernel` is called to fill in the score variable after an all-to-all pair sequence alignment. Note the inverted loop structure from Algorithm 3.

```
1 for t in T:
2   score = [i16(0) for i in range(len(t))]
3   t_vec = [Vec(ord(str(t[i])), 16, dtype=u8)
4             for i in range(len(t))]
5   for q in Q:
6     q_vec = [[Vec(ord(str(q[i][j])), 16, dtype=u8)
7               for i in range(len(Q))]
8              for j in range(len(q))]
9     yield simd_kernel(t_vec, q_vec, Mp, *args)
```

Note that a naïve construction of this 3D structure can result in large memory overhead and can have detrimental impacts on cache performance. To address these issues, Vectron employs a technique called **double buffering**. This technique allows for efficient memory usage by maintaining only a small subset of data in active memory while minimizing the frequency of costly data transfers.

Specifically, Vectron reorders the `for` loops in the scheduler and the recurrence block in a cache-oblivious manner to reduce cache misses during dynamic programming (DP) operations. As seen in Algorithm 3, Vectron inverts the two middle loops, fixing a single target t for comparison against all queries q. This structure allows for the reuse of values in the scoring matrix, retaining only the necessary data (e.g., a previous row in the case of Needleman-Wunsch) while replacing outdated information in a rolling fashion.

By implementing this double-buffering scheme, Vectron avoids costly data transfers and cache invalidations, leading to improved performance. As illustrated in Figure 4.1, our findings indicate that this approach yields an average performance improvement of 30% in C++ and a 12% boost when using Vectron compared to the naïve data-structure approach.

Algorithm 3 The inverted loop structure processes all elements of Q for each element $t[i] \in t$ in T, exposing opportunities for SIMD vectorization and parallelization. Each comparison of fixed $t[i]$ with every $q[j]$ in q can be independently executed with increased cache performance.

```
1:  Input: Sets of sequences T and Q
2:  Part 1: The standard nested DP loop structure.
3:  for each t in T do
4:      for each q in Q do
5:          for i in 0...len(t) − 1 do
6:              for j in 0...len(q) − 1 do
7:                  f(t[i], q[j])
8:              end for
9:          end for
10:     end for
11: end for
12: Part 2: The inverted DP loop structure.
13: for each t in T do
14:     for i in 0...len(t) − 1 do
15:         for each q in Q do
16:             for j in 0...len(q) − 1 do
17:                 f(t[i], q[j])
18:             end for
19:         end for
20:     end for
21: end for
```

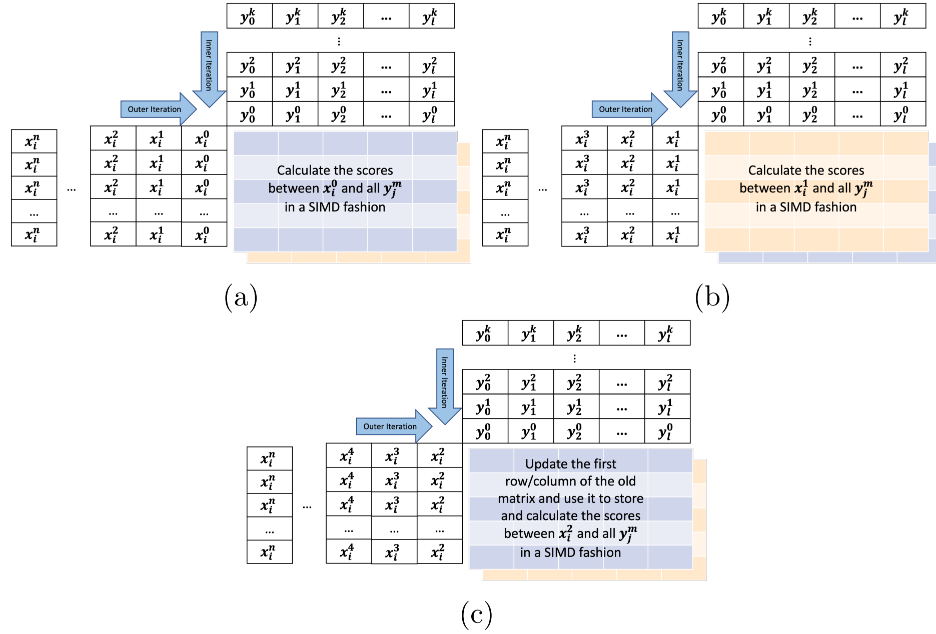


Figure 4.1: A cache-friendly 3D matrix structure and the inverted loop strategy that utilizes double buffering. (a) The initial iteration compares x_i^0 with all elements y_j, resulting in a scoring matrix that is inserted into the 3D structure. (b) The second iteration compares x_i^1 with all y_j, generating another matrix that is appended to the 3D structure. (c) The third iteration follows the same pattern as the previous iterations. However, in this case, the new matrix replaces the one from the first iteration in a rolling fashion. This mimics a double-buffering scheme, where the outdated matrix is discarded as it will not be needed later on. This technique helps maintain a minimal memory footprint while maximizing cache efficiency.

4.5 GPU support

If a user wishes to target GPUs instead of CPUs via SIMD intrinsics, Vectron will utilize Codon's GPU support Shajii et al. [2023] and will decorate the main loops with `@par(gpu=True)` decorator to let the Codon know that the loops will be parallelized through GPU. Codon will then automatically convert that Vectron kernel into a CUDA-compatible kernel via NVPTX. Vectron will also infer the GPU parameters after the code analysis (e.g., the `collapse` variable that lets Codon know how many nested loops should be collapsed into GPU threads).

Chapter 5

Experiments

We evaluated Vectron on a large set of dynamic programming problems and datasets. The benchmarks were run on a 24-core Intel Xeon 8260 CPU (2.40 GHz) machine with 1 TB of DDR4 RAM and an NVIDIA Tesla V100 GPU with 32 GB of VRAM. The available SIMD instruction sets included SSE 4.1, SSE 4.2, AVX, AVX2 and AXV-512. All runs were done on a 64-bit CentOS 7 Linux. We used Codon v0.16.3 and LLVM v15, as well as gcc 12.3 and clang 17 for the builds. All runs were compiled with optimizations (`-O2` or `-O3` for C++ and `-release` for Codon). Compile times are not included in the overall run.

5.1 Experiment Scenarios

We conducted experiments using different datasets to evaluate the performance of the dynamic programming algorithms under different loads. The experiments involved comparing different numbers of target and query sequence pairs with a fixed length of 512 for each sequence. These experiments were divided into three sets: small (4,096 sequence pairs), medium (262,144 sequence pairs) and large (4,194,304 sequence pairs). Each experiment was run 5 times, and for each experiment, we reported the average over all runs.

For each dynamic programming algorithm, we implemented three versions: a baseline C++ version, a baseline Python/ Codon version and a Vectron-enabled Codon version. A baseline version consists of a textbook implementation of a single instance DP kernel and a simple sequential loop that invokes the kernel for each available instance (pair). The C++ implementations were then compiled using `clang++ -O3`

`-msse4.2 -funroll-loops -mfpmath=sse -march=native`. (The same implementations were compiled using `clang++ -mllvm --polly` as well, and no improvements were observed compared to the first compilation method). The Vectron-enabled version is the same as the corresponding baseline version except for the addition of a `@vectron` decorator.

For a select subset of algorithms, we also compared Vectron implementations with the state-of-the-art solutions. In the case of highly optimized single-instance DP kernels, we compared Vectron with a loop that calls the optimized kernel for each instance. These cases include kernels from Parasail (Daily [2016]), SSW (Zhao et al. [2013]) and AAlign (Hou et al. [2016]). Other methods, such as Seq (Shajii et al. [2019, 2021]) and SWIPE (Rognes [2011]), support inter-alignment (multi-instance) workflows; in these cases, we compared Vectron directly against the respective vectorized inter-alignment version. Note that we did not directly compare against KSW2 (Li [2020])—one of the fastest SIMD implementations of Smith-Waterman—because it is already used internally by Seq. Additionally, Codon employs Seq for all Smith-Waterman variations.

Unless otherwise noted, we used the most complex Smith-Waterman variant with an affine gap penalty and Gotoh scoring (with both integer and floating-point penalties) as the reference benchmark because it includes all corner cases and encapsulates all other variants. All scores are assumed to fit into a 16-bit integer. A list of other supported DP algorithms and their variants is listed in Table 5.1. Finally, note that we were unable to compare Vectron against other DP methods (e.g., XLCS for LCS) due to the unavailability of their source code.

5.2 CPU (SIMD) Results

We first compared Vectron implementations targeting SIMD against the baseline C++ and Codon variants. For our SIMD setup, we used AVX2 to be able to accommodate vectors of length 16 containing 16-bit integers. As seen in Table 5.2, Vectron achieves an average speedup of 45× over Codon's sequential implementation and an average speedup of 18× over C++ implementations. Note that the performance difference keeps increasing as the number of pairs keeps increasing. This performance speedup trend is also illustrated in Figure 5.1 for different algorithms and different dataset sizes.

All Vectron code is implemented as a simple set of DP loops without any man-

Table 5.1: Supported algorithms by the evaluated state-of-the-art methods for Smith-Waterman calculation. Other tools were not evaluated as their source code was unavailable.

Tool	Supported Algorithms
Vectron	Smith-Waterman (all variations), Needleman-Wunsch, Manhattan Tourist, LCS, Levenshtein Distance, Minimum Cost Path, Hamming Distance, Optimal Binary Search Tree, Longest Increasing Subsequence, etc.
Seq	Smith-Waterman (all variations)
SSW	Smith-Waterman (all variations)
SeqAn	Smith-Waterman, Needleman-Wunsch (all variations)
SWIPE	Smith-Waterman (all variations)
Parasail	Smith-Waterman (all variations), Needleman-Wunsch
AAlign	Smith-Waterman (all variations), Needleman-Wunsch

ual intervention except for the addition of `@vectron` decorator. Code samples are available in the Appendix.

Next, we compared Vectron on a small dataset (4,096 sequence pairs) against single-instance state-of-the-art SIMD methods (Parasail Daily [2016], SSW Zhao et al. [2013] and Aalign Hou et al. [2016]), and multi-instance state-of-the-art methods (Seq's `@inter_align` alignment Shajii et al. [2019, 2021], SeqAn Rahn et al. [2018] and SWIPE Rognes [2011]). All tools were built on the same hardware as Vectron. Unfortunately, the source code for other implementations (e.g., Bednárek et al. [2015], Jararweh et al. [2019], Yin et al. [2019]) was not available at the time of writing this thesis. In addition to elapsed time, we also provide measure performance metrics in GCUPS (Giga Cell Updates per Second) originally proposed by Daily [2016]. More specifically, if we denote the number of matrices used in the algorithm as n, the size of each of these matrices as l (assuming all matrices are $l \times l$), the number of sequence pairs as p, and the run-time of the algorithm as t in seconds, then the speed in GCUPs can be calculated as:

$$\text{GCUPS} = \frac{n \times l \times p}{t}.$$

The results of this benchmark are shown in Table 5.3. As can be seen there, the Vectron-optimized version surpasses all other tools from 10% to more than 10×.

Finally, we compared Vectron against baseline C++ versions vectorized by RV vectorization suite Schryver et al. [2024]. The results for different DP algorithms on small datasets are illustrated in Table 5.4. As can be seen, RV is not able to sufficiently

Table 5.2: Comparison of SIMD-vectorized Vectron DP implementations and the equivalent baseline C++ and Codon versions on a small set of 4,096 sequence pairs (top), medium set of 262,144 sequence pairs (middle), and a large set of $4,194,304$ sequence pairs (bottom). Each sequence in a pair is of length 512. All times are shown in seconds (lower is better). The best times are highlighted in boldface. We also show Vectron's speed-up factor next to each other implementation (higher is better).

Algorithm	Vectron	Codon	Speed-up	C++	Speed-up
Smith-Waterman	**0.27 ± 0.01**	0.30 ± 0.01	1.11×	5.78 ± 0.17	21.41×
Needleman-Wunsch	**0.45 ± 0.01**	13.32 ± 0.40	29.63×	6.18 ± 0.19	13.73×
Levenshtein Distance	**0.34 ± 0.01**	13.25 ± 0.40	38.97×	4.88 ± 0.15	14.35×
Longest Common Subsequence	**0.32 ± 0.01**	13.20 ± 0.39	41.25×	4.16 ± 0.12	13.00×
Hamming Distance	**0.07 ± 0.00**	4.90 ± 0.15	70.00×	1.45 ± 0.04	20.71×
Manhattan Tourist	**0.51 ± 0.02**	12.30 ± 0.37	24.12×	4.91 ± 0.15	9.63×
Minimum Cost Path	**0.40 ± 0.01**	17.70 ± 0.53	44.25×	6.05 ± 0.18	15.13×
Average Speedup			35.61×		15.42×

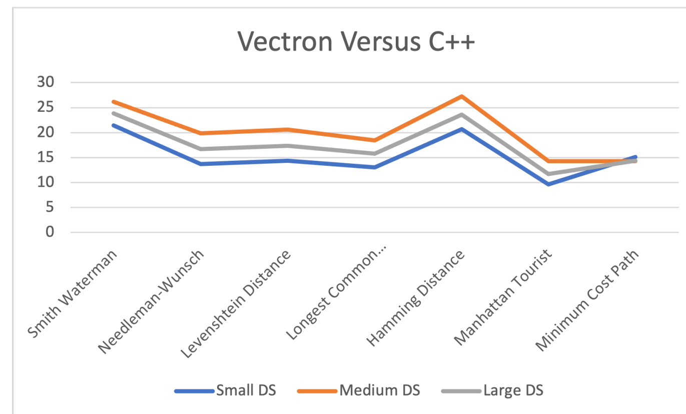
Algorithm	Vectron	Codon	Speed-up	C++	Speed-up
Smith-Waterman	**15.12 ± 0.45**	19.29 ± 0.58	1.28×	395.93 ± 11.88	26.19×
Needleman-Wunsch	**23.89 ± 0.72**	841.4 ± 25.24	35.21×	474.66 ± 14.24	19.87×
Levenshtein Distance	**17.84 ± 0.54**	746.83 ± 22.40	41.86×	367.58 ± 11.03	20.60×
Longest Common Subsequence	**16.54 ± 0.50**	855.55 ± 25.67	51.73×	305.24 ± 9.16	18.45×
Hamming Distance	**2.27 ± 0.07**	286.76 ± 8.60	126.33×	61.92 ± 1.86	27.28×
Manhattan Tourist	**24.30 ± 0.73**	771.84 ± 23.16	31.76×	347.23 ± 10.42	14.29×
Minimum Cost Path	**20.42 ± 0.61**	1043.89 ± 31.32	51.12×	291.76 ± 8.75	14.29×
Average Speedup			48.47×		20.14×

Algorithm	Vectron	Codon	Speed-up	C++	Speed-up
Smith-Waterman	**235.63 ± 7.07**	306.36 ± 9.19	1.30×	5620.19 ± 168.61	23.85×
Needleman-Wunsch	**341.00 ± 10.23**	12401.50 ± 372.05	36.37×	5691.12 ± 170.73	16.69×
Levenshtein Distance	**269.07 ± 8.07**	11413.90 ± 342.42	42.42×	4682.71 ± 140.48	17.40×
Longest Common Subsequence	**246.35 ± 7.39**	11287.10 ± 338.61	45.82×	3876.42 ± 116.29	15.74×
Hamming Distance	**31.30 ± 0.94**	4460.73 ± 133.82	142.52×	739.07 ± 22.17	23.61×
Manhattan Tourist	**364.62 ± 10.94**	10402.20 ± 312.07	28.53×	4259.60 ± 127.79	11.68×
Minimum Cost Path	**304.80 ± 9.14**	16463.50 ± 493.91	54.01×	4381.37 ± 131.44	14.37×
Average Speedup			50.14×		17.62×

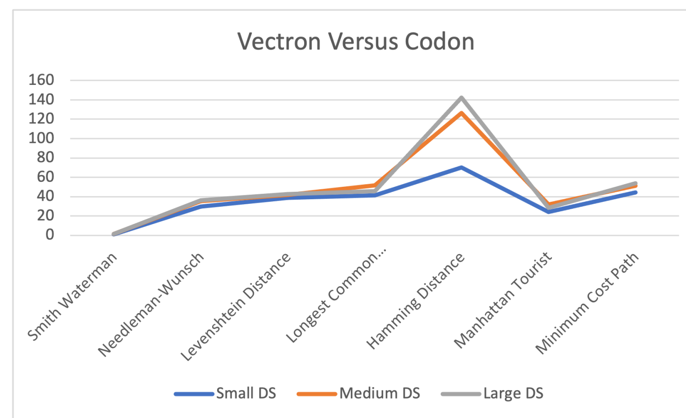
auto-vectorize any of the dynamic programming algorithms, and its versions are, on average, more than 100 times slower than Vectron.

5.3 GPU Results

To benchmark the performance of Vectron on GPUs, we evaluated the performance of the Gotoh Smith-Waterman algorithm with a 32-bit floating-point scoring scheme. The Vectron GPU backend generated the GPU-compatible Codon code that was



(a) Speedup of Vectron versus C++ for various algorithms and dataset sizes.



(b) Speedup of Vectron versus Codon for various algorithms and dataset sizes.

Figure 5.1

Table 5.3: Performance of the state-of-the-art algorithms. The best values are highlighted in boldface. The methods are grouped depending if they are optimized for single-instance or multi-instance DP.

Algorithm	GCUPS (± Std. Dev.)	Time (± Std. Dev.)
SSW	9.24 (± 0.15)	0.35 (± 0.02)
Parasail	6.54 (± 0.12)	0.49 (± 0.03)
AAlign	0.01 (± 0.00)	294.00 (± 1.20)
Seq `@inter_align`	10.77 (± 0.18)	0.30 (± 0.01)
SeqAn	4.97 (± 0.10)	0.65 (± 0.04)
SWIPE	1.25 (± 0.05)	2.58 (± 0.10)
Vectron	**13.24** (± 0.20)	**0.27** (± 0.01)

Table 5.4: Runtime comparison of Vectron against RV on small (4,096 sequence pairs) dataset. All numbers are reported in seconds.

Algorithm	Vectron (± Std. Dev.)	C++ (RV) (± Std. Dev.)
Smith-Waterman	**0.27** (± 0.01)	39.59 (± 0.80)
Needleman-Wunsch	**0.45** (± 0.01)	65.10 (± 1.30)
Levenshtein Distance	**0.34** (± 0.01)	63.91 (± 1.20)
LCS	**0.32** (± 0.01)	61.74 (± 1.10)
Hamming Distance	**0.07** (± 0.00)	6.55 (± 0.20)
Manhattan Tourist	**0.51** (± 0.02)	61.92 (± 1.15)
Minimum Cost Path	**0.40** (± 0.01)	143.68 (± 2.50)

compiled to CUDA kernels via Codon's `gpu` library and LLVM's NVPTX module. We also implemented the same inter-alignment kernels manually in C++ with NVIDIA CUDA's C++ library and NVCC 12 compiler. For these benchmarks, we used 256, 1,024, 4,096, 16,384, and 65,536 sequence pairs of length 512 each. Larger datasets could not fit our GPU memory and were thus omitted.

The results in Table 5.5 show the comparison of Vectron's GPU code over the equivalent C++ implementations. As can be seen, both GPU implementations are slightly slower on small loads due to the cost of memory transfer to the GPU. However, as the size of the dataset increases, the memory transfer costs are offset, and the GPU versions become much faster.

It is important to note that Codon's GPU support is still under heavy development and that there are some limitations in the current version. For example, manual C++ CUDA is faster on 4,096 pairs due to the ability to select a more efficient memory allocation strategy. While this strategy is currently unsupported by Codon, we note that it fails on larger loads. We expect this issue to be resolved in the upcoming Codon versions. Nevertheless, it can be seen that with the addition of a single line (`@par(gpu=True)`), any matrix-based DP algorithm can be effortlessly executed on GPU with a good performance.

We also compared both SIMD and GPU-enabled Vectron versions of Smith-Waterman against the state-of-the-art GPU Smith-Waterman implementations. The results are shown in Table 5.6. While Vectron's GPU version demonstrates slower performance compared to ADEPT (Awan et al. [2020]), a leading implementation for DP algorithms on GPUs, its SIMD version is marginally faster than ADEPT when aligning the small dataset of 4,096 sequence pairs. Similarly, SW# (Okada et al.

Table 5.5: GPU performance comparison between Vectron and C++ versions. Both Vectron and CUDA are slower than the baseline C++ version on small loads due to the cost of copying values between the GPU and the main memory.

Number of Pairs ↓	Execution Time (s) (± Std. Dev.)		
	Vectron	C++ (CUDA)	C++ (baseline)
256	0.42 (± 0.02)	1.83 (± 0.10)	0.25 (± 0.01)
1024	1.31 (± 0.05)	1.95 (± 0.12)	1.05 (± 0.03)
4096	5.15 (± 0.20)	2.35 (± 0.15)	6.25 (± 0.25)
16384	23.3 (± 0.90)	Out of Memory	44.32 (± 1.50)
65536	105.63 (± 3.50)	Out of Memory	205.18 (± 7.00)

[2015]) operates under similar conditions as ADEPT, running in the integer domain and is slower more than 6× than ADEPT or Vectron's SIMD version. It does, however, outperform Vectron's GPU version. The discrepancy between Vectron's GPU and SIMD versions is due to the fact that Vectron's GPU backend currently spends considerable time converting all input integers into floats. This is also one of the things expected to be resolved in the upcoming updates to Codon's GPU backend.

Table 5.6: Performance comparison between state-of-the-art GPU implementations and Vectron's SIMD and GPU implementations of Smith-Waterman with integer scoring.

Algorithm	Execution Time (s) (± Std. Dev.)
Vectron CPU	**0.27** (± 0.01)
Vectron GPU	5.15 (± 0.20)
ADEPT Awan et al. [2020]	0.28 (± 0.02)
SW# Okada et al. [2015]	1.77 (± 0.10)

Chapter 6

Conclusion

In this thesis, we presented Vectron, a Codon compiler auto-vectorization pass that transforms array-based and Pythonic dynamic programming implementations to highly optimized vectorized counterparts that can efficiently process multiple instances in parallel. Vectron relies upon Codon's SIMD and GPU capabilities to achieve this. We show that Vectron can achieve more than 20× speedup over the baseline C++ versions or even over the highly optimized, custom-tailored solutions, without necessitating algorithm changes or large-scale code refactoring. Thus, we hope that Vectron will be a useful tool for people who need to efficiently solve many instances of dynamic programming algorithms.

Future work on the Vectron will focus on leveraging multithreading and cluster environments in both CPU and GPU implementations to further enhance its performance and scalability. We also plan to support vectorization of a larger set of allowed expressions, as well as improved scheduling capabilities for streaming use cases where the size of the data is not known in advance. Finally, we plan to extend the underlying GPU support to support more complex use cases and memory allocation schemes.

Bibliography

0xTCG. Vectron: A dynamic programming auto-vectorization framework. `https://github.com/0xTCG/vectron/`, 2024. Accessed: 2024-08-06.

Bowen Alpern, Larry Carter, and Kang Su Gatlin. Microparallelism and high-performance protein matching. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pages 24–es, New York, NY, USA, 1995. ACM/IEEE.

Stephen F. Altschul. Generalized affine gap costs for protein sequence alignment. *Proteins: Structure, Function, and Bioinformatics*, 32(1):88–96, 1998.

Srinivas Aluru. *Handbook of Computational Molecular Biology*. Chapman and Hall/CRC, 2005.

Muaaz G. Awan et al. Adept: a domain-independent sequence alignment strategy for gpu architectures. *BMC Bioinformatics*, 21:1–29, 2020.

Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 51–58. ACM, 2015. ISBN 978-1-4503-3536-2. doi: 10.1145/2746539.2746612.

Richard Barnes. A review of the smith-waterman gpu landscape. *Berkeley: Electrical Engineering and Computer Sciences, University of California*, Volume Number (if applicable)(Issue Number (if applicable)):1–23, 2020.

David Bednárek, Michal Brabec, and Martin Kruliš. On parallel evaluation of matrix-based dynamic programming algorithms. 2015.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. Pluto: An automatic parallelizer and locality optimizer. `https://pluto-compiler.sourceforge.net`, 2008. Accessed: 2024-08-06.

Jeff Daily. Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17(1):1–11, 2016.

Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics*, 9:1–9, 2008.

M. Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007.

Zvi Galil and Kunsoo Park. A linear-time algorithm for concave one-dimensional dynamic programming. 1989.

O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

Kaixi Hou, Hao Wang, and Wu-chun Feng. Aalign: A simd framework for pairwise sequence alignment on x86-based multi-and many-core processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 780–789. IEEE, 2016.

Yaser Jararweh et al. Improving the performance of the needleman-wunsch algorithm using parallelization and vectorization techniques. *Multimedia Tools and Applications*, 78:3961–3977, 2019.

Jaeho Kim, Byunggab Kim, and Sungroh Kim. Dynamic programming-based multiple sequence alignment for high-throughput dna sequencing. *Bioinformatics*, 34(20): 3437–3445, 2018. doi: 10.1093/bioinformatics/bty380.

Simon Knee. *Opal*. PhD thesis, University of Oxford, 1997.

J. B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, 1983.

Chris Lattner and Vikram Adve. Llvm: An infrastructure for multi-stage optimization. In *Proceedings of the 2002 International Symposium on Code Generation and Optimization*, pages 75–88. IEEE Computer Society, 2002.

Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34 (18):3094–3100, 2018.

Heng Li. Ksw2: Algorithm for optimal alignment between similar sequences. *Bioinformatics*, 36(7):2165–2171, 2020. doi: 10.1093/bioinformatics/btz859.

Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. doi: 10.1093/bioinformatics/btp324. URL `https://academic.oup.com/bioinformatics/article/25/14/1754/225615`.

Yi-Lun Liao, Yi-Chang Li, Nai-Cheng Chen, and Yu-Chen Lu. Adaptively banded smith-waterman algorithm for long reads and its hardware accelerator. *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–9, 2018.

Xue Liu, Jiaheng Wu, and Youmin Gong. Scalable many-to-many sequence alignment using divide-and-conquer dynamic programming. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1319–1331, 2019. doi: 10.1109/TPDS.2018.2870144.

W. Miller and E. W. Myers. Sequence comparison with concave weighting functions. *Bulletin of Mathematical Biology*, 50(2):97–120, 1988.

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

S. B. Needleman and C. D. Wunsch. A general method, applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

Daiki Okada, Fumihiko Ino, and Kenichi Hagihara. Accelerating the smith-waterman algorithm with interpair pruning and band optimization for the all-pairs comparison of base sequences. *BMC Bioinformatics*, 16:1–15, 2015.

René Rahn, Stefan Budach, Pascal Costanza, Marcel Ehrhardt, Jonny Hancox, and Knut Reinert. Generic accelerated sequence alignment in seqan using vectorization and multi-threading. *Bioinformatics*, 34(20):3437–3445, 2018. doi: 10.1093/bioinformatics/bty380. URL `https://doi.org/10.1093/bioinformatics/bty380`.

T. Rognes. Faster smith-waterman database searches with inter-sequence simd parallelization. *BMC Bioinformatics*, 12(1):221, 2011.

W. R. Rudnicki, A. Jankowski, A. Modzelewski, A. Piotrowski, and A. Zadrożny. The new simd implementation of the smith-waterman algorithm on cell microprocessor. *Fundamenta Informaticae*, 96(1-2):181–194, 2009.

D. Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6, 1972.

Christian De Schryver et al. Rv: A unified region vectorizer for llvm, 2024. URL `https://github.com/cdl-saarland/rv`. GitHub repository.

Sean O. Settle. High-performance dynamic programming on fpgas with opencl. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jaewoo Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. Codon: A compiler for high-performance pythonic applications and dsls. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 191–202, New York, NY, USA, February 17 2023. ACM.

Ariya Shajii et al. Seq: a high-performance language for bioinformatics. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

Ariya Shajii et al. A python-based programming language for high-performance computational genomics. *Nature Biotechnology*, 39(9):1062–1064, 2021.

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

T. Stoyanova, V. Leung, and S. Yalamanchili. Parallel sequence comparison with the smith-waterman algorithm using mic architecture. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium Workshops*, 2014.

Hajime Suzuki and Masahiro Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, 19:33–47, 2018.

Yufeng Wang, Xue Liu, Li Xie, Kui Liu, Xiangning Li, Yong Liu, et al. Parallelizable and memory-efficient dynamic programming algorithms for multiple sequence

alignment. *Bioinformatics*, 37(17):2534–2541, 2021. doi: 10.1093/bioinformatics/btab157.

M. S. Waterman. Efficient sequence alignment algorithms. *Journal of Theoretical Biology*, 108(3):333–337, 1984.

M. S. Waterman, T. F. Smith, and W. A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976.

F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the 12 Annual ACM Symposium on Theory of Computing (STOC'80)*, pages 429–43, 1980.

Zekun Yin, Hao Zhang, Kai Xu, Yuandong Chan, Shaoliang Peng, Xiaoning Wang, Bertil Schmidt, and Weiguo Liu. Xlcs: A new bit-parallel longest common subsequence algorithm on xeon phi clusters. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1477–1483. IEEE, 2019.

Yuchao Zhang, Yiming Li, and Heng Huang. Efficient many-to-many sequence alignment using progressive dynamic programming. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 17(6):1939–1951, 2020. doi: 10.1109/TCBB.2019.2916715.

Mengyao Zhao, Wan-Ping Ng, Saeed Rahmann, Benno Schwikowski, Alex Bateman, Sarah Teichmann, Erik Lindahl, Jennifer Harrow, and Erik Sonnhammer. Ssw library: an simd smith-waterman c/c++ library for use in genomic applications. *PloS One*, 8(12):e82138, 2013.

Martin Šošić and Mile Šikić. Edlib: a c/c++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

Appendix A

Additional Information

A.1 Artifact Appendix

A.1.1 Abstract

The artifact described in 0xTCG [2024] includes a `docker` folder containing three Dockerfiles. It implements an auto-vectorization approach for many-targets to many-queries DP implementations called Vectron and other state-of-the-art implementations presented in the thsis. The artifact requires NVIDIA (CUDA)-enabled hardware for proper execution.

A.1.2 Artifact check-list (meta-information)

- **Algorithm:** Auto-vectorization of many-to-many DP algorithms
- **Program:** 5 datasets of DNA sequence pairs (including the ambiguous nucleotide). All present, or created during Dockerization.
- **Compilation:** LLVM, clang++ (3 versions), Codon, Seq, Vectron, SSW, Parasail, SeqAn, Adept, Swipe, SW#, Aalign and RV packages are required to run the entire experiment set presented in the thesis. All packages will be built and installed during Dockerization. podman 4.9.5 was used and is recommended for the Dockerization process.
- **Binary:** All required binaries will be built during Dockerization.
- **Data set:** refer to A.1.2 for details on datasets.

- **Run-time environment:** The first two docker images are built using nvidia/cuda:12.4.1–devel–ubuntu22.04 for all experiments excluding Aalign and SW#. These two experiments are built in the last docker image using nvidia/cuda:11.6.1–devel–ubuntu20.04.

- **Hardware:** The benchmarks were run on a 24-core Intel Xeon 8260 CPU (2.40 GHz) machine with 1 TB of DDR4 RAM and an NVIDIA Tesla V100 GPU with 32 GB of VRAM. The available SIMD instruction sets included SSE 4.1, SSE 4.2, AVX, AVX2 and AXV-512. All runs were done on a 64-bit CentOS 7 Linux. For experimenting with Codon and Vectron we used Codon v0.16.3 and LLVM v15, as well as gcc 12.3, clang 17, and cuda 12.0.4 for the builds. Other LLVM and clang builds and versions were also used for implementing other state-of-the-art implementations. podman 4.9.5 was also used for the Dockerization process. The nvidia/cuda:12.4.1–devel–ubuntu22.04 environment was also tested comprehensively during the Dockerization process, and any NVIDIA(CUDA)-enabled system that can create a nvidia/cuda:12.4.1–devel–ubuntu22.04 and a nvidia/cuda:11.6.1–devel–ubuntu20.04 docker containers should be theoretically capable of performing all experiments presented in the thesis.

- **Execution:** First build the vectron, and then the others, and lastly the cuda11 containers. To benchmark Vectron run the vectron container, otherwise run others for other state-of-the-art implementations except for Aalign and SW#. Run cuda11 to benchmark Aalign and SW#. Although speedups will not change for Vectron while running others, the actual runtimes might increase, and therefore running vectron individually would save some evaluation process time.

- **Metrics:** Execution Time.

- **Output:** The output of all experiments is the alignment score between all the Query and Target sequence pairs which is stored in files, and the Execution Time of each experiment.

- **Experiments:** Building and running the Dockerfiles using the instructions in this appendix will create two final .ipynb files: benchmarks.ipynb and others_benchmarks.ipynb. Each Docker container will run a jupyter–lab server which can be accessed to observe/run the experiments. Running the cuda11 container will bring up a bash environment to benchmark Aalign and SW#.

- **How much disk space required (approximately)?:** Around 15 to 20 GB.
- **How much time is needed to prepare workflow (approximately)?:** The Docker images will be built in less than 90 minutes on a decent system.
- **How much time is needed to complete experiments (approximately)?:** The experiments will take around 150 minutes for everything except Codon and C++ experiments for the large dataset in CPU mode. The Codon experiment for the large dataset in the CPU mode takes around 30 hours and the C++ experiment for the large dataset in CPU mode takes around 12 hours.
- **Publicly available?:** All repositories used in the experiments are publicly available (including Codon and Vectron).

A.1.3 Description

How delivered

The artifact evaluation package can be cloned from Vectron on GitHub. The docker folder will have all the required Dockerfiles and misc. files to reproduce the experiments.

Hardware dependencies

The most important dependency is the availability of an NVIDIA GPU (with CUDA version 11.0 and above).

Software dependencies

A UNIX-based or a MAC-based system.

Data sets

The datasets are all generated and non-private. They either already exist in the docker folder, or get created during the Dockerization process. The Small Int, Large Float and others datasets are identical.

A.1.4 Installation

After cloning the repository follow these steps:

- If you are connecting to a remote server to run the experiments, do so with port-mapping 8888:8888. For instance: ssh –L 8888:localhost:8888 user@server

- Navigate to the \$PWD\$\\vectron folder and build the corresponding Docker container using a sample command: podman build ––security–opt label=disa ble ––security–opt seccomp=unconfined ––tag vectron .

- the vectron container can now be run using a run command with –p 8888:8888 ––device nvidia.com/gpu=all ––security–opt label=disable ––security–opt s eccomp=unconfined . A sample command would be: podman run –it –p 8888: 8888 ––device nvidia.com/gpu=all ––security–opt label=disable ––security –opt seccomp=unconfined vectron

- To build the experiments for other state-of-the-art implementations on the host system, navigate to the \$PWD\$\\others folder and build the others Docker container in the same manner done in A.1.4. A sample command for this would be: podman build ––security–opt label=disable ––security–opt seccomp=un confined ––tag vectron .

- others container can now be run using the same approach in A.1.4. A sample command for this would be podman run –it –p 8888:8888 ––device nvidia.co m/gpu=all –– security–opt label=disable ––security–opt seccomp=unconfi ned others

- To build the Aalign and SW# experiments on the host system, navigate to the \$PWD\$\\cuda11 folder and build the cuda11 Docker container in the same manner done in A.1.4. A sample command for this would be: podman build –– security–opt label=disable ––security–opt seccomp=un confined ––tag cud a11 .

- cuda11 container can now be run using the same approach in A.1.4. A sample command for this would be podman run –it –p 8888:8888 ––device nvidia.co m/gpu=all –– security–opt label=disable ––security–opt seccomp=unconfi ned cuda11

- it is recommended to run the vectron container for Vectron, Codon and C++ experiments, and the others container for benchmarking other state-of-the-art implementations. Although both experiment environments will be available via the others environments and the relative speedups remain unchanged for Vectron, Codon, and C++ implementations, the individual runtimes for these experiments have been considered to be higher when run in the vectron container.

- After either vectron or others container is run and loaded with the jupyter lab server, copy the token value by copying the value after the phrase ?token in the output log of the jupyter lab server. For instance in http://127.0.0.1:8888/lab?token=865f18716a91a2bd3e985ba68c 9e780ad7354b21c8198 3d6, the token value would be 865f18716a91a2bd3e98 5ba68c9e780ad7354b21c81983d6.

- Inside any browser on the host system browse to this address: http:// localhost :8888/?token=[TOKEN_ID]. This would load the jupyter lab environment for the experiments.

- After the cuda11 is run Aalign and SW# benchmarks can be run in the newly created bash environment using these commands: time /swsharp/bin/swsharpdb –i /others/data/seqx.fasta –j /others/data/seqy.fasta and time /aalign/ModularDesign/Smi thWatermanAffineGapMod.out –q /others/data/seqx.fasta –d /others/data/seqy.fasta

A.1.5 Experiment workflow

Each of the benchmarks.ipynb or others_benchmarks.ipynb notebooks have full descriptions of the experiment workflow for Vectron, Codon, C++ and other state-of-the-art implementations.

A.1.6 Evaluation and expected result

The expected results are shown in the .ipynb files during previous runs. All experiments can be evaluated by running the cells inside the notebooks again (except for Aalign and SW#).

A.1.7 Experiment customization

All the Vectron (Codon), C++ and CUDA scripts are available in PWD
docker
experiments_docker
source. They can be modified to run customized experiments.

A.2 Sample Source Code Appendix

This section contains sample implementations of popular DP algorithms that are auto-vectorized by Vectron.

Code Block 8 Smith-Waterman with Gotoh scoring (affine-gap penalties).

```
1  @vectron.cmp
2  def S(q, t, match, mismatch):
3    return match if q == t else mismatch
4  @vectron
5  def gotoh(q, t, gap_o, gap_e, match, mismatch):
6    m, n = len(q), len(t)
7    # Initialize matrices E, F, M with dimensions (len(t)+1) * (len(q)+1)
8    # with zeroes
9    # Initialize the first row of M, F, and H with gap_o + row_i * gap_e
10   # Initialize the first column of M, E, and H with gap_o + column_j *
11   # gap_e
12   M = [[0] * (len(q) + 1) for _ in range(len(t) + 1)]
13   E = [[0] * (len(q) + 1) for _ in range(len(t) + 1)]
14   F = [[0] * (len(q) + 1) for _ in  range(len(t) + 1)]
15   for i in range(len(t) + 1):
16     M[i][0] = gap_o + i * gap_e
17     E[i][0] = gap_o + i * gap_e
18   for j in range(len(q) + 1):
19     M[0][j] = gap_o + j * gap_e
20     E[0][j] = gap_o + j * gap_e
21
22   # Run DP
23   for i in range(1, len(t)+1):
24     for j in range(1, len(q)+1):
25       E[i][j] = max(
26         E[i][j-1] + gap_e,
27         H[i][j-1] + gap_o + gap_e
28       )
29       F[i][j] = max(
30         F[i-1][j] + gap_e,
31         H[i-1][j] + gap_o + gap_e
32       )
33       M[i][j] = max(
34         E[i][j],
35         F[i][j],
36         M[i-1][j-1] + S(
37           q[j-1], t[i-1], match, mismatch
38         )
39       )
40
41   # Alignment score between Q and T is M[len(t)][len(q)]
42   return M[len(t)][len(q)]
```

Code Block 9 Hamming Distance calculation.

```
1 @vectron
2 def hamming(q, t, match, mismatch):
3   m, n = len(q), len(t)
4   M = [[0] * (m + 1) for _ in range(n + 1)]
5   for i in range(1, n+1):
6     for j in range(1, m+1):
7       M[i][j] = max(
8         M[i-1][j-1] + (match if q[j-1] == t[i-1] else mismatch),
9         0
10      )
11  return M[n][m]
```

Code Block 10 – Levenshtein (Edit) Distance algorithm.

```
1 @vectron.cmp
2 def S(q, t, match, mismatch):
3   return match if q == t else mismatch
4 @vectron
5 def levenshtein(Q, T, gap, match, mismatch)
6   # Initialize M with dimensions (n+1) * (m+1) with zeroes
7   # Initialize the first row of M with row_i * gap
8   # Initialize the first column of M with column_j * gap
9   M = [[0] * (m + 1) for _ in range(n + 1)]
10  for i in range(n + 1):
11    M[i][0] = gap_o + i * gap_e
12  for j in range(m + 1):
13    M[0][j] = gap_o + j * gap_e
14
15  for i in range(1, n+1):
16    for j in range(1, m+1):
17      M[i][j] = max(
18        M[i - 1][j] + gap,
19        M[i][j - 1] + gap,
20        M[i-1][j-1] + S(q[j-1], t[i-1], match, mismatch)
21      )
22  return M[n][m]
```

Code Block 11 Longest Common Subsequence (LCS) algorithm.

```
1 @vectron
2 def lcs(q, t, match, mismatch)
3   # Initialize M with dimensions (n+1) * (m+1) with zeroes
4   M = [[0] * (m + 1) for _ in range(n + 1)]
5   for i in range(1, n+1):
6     for j in range(1, m+1):
7       M[i][j] = max(
8         M[i-1][j],
9         M[i][j-1],
10        M[i-1][j-1] + (match if q[j-1] == t[i-1] else mismatch)
11      )
12  return M[n][m]
```

Code Block 12 Manhattan Tourist algorithm.

```
1 @vectron.cmp
2 def S(q, t, match, mismatch):
3   return match if q == t else mismatch
4 @vectron
5 def manhattan(q, t, match_1, mismatch_1,  match_2, mismatch_2)
6   # Initialize M with dimensions (n+1) * (m+1) with zeroes
7   M = [[0] * (m + 1) for _ in range(n + 1)]
8
9   for i in range(1, n+1):
10    for j in range(1, m+1):
11      M[i][j] = max(
12        M[i][j-1] + S(q[j-1], t[i-1], match_1, mismatch_1),
13        M[i-1][j] + S(q[j-1], t[i-1], match_2, mismatch_2)
14      )
15  return M[n][m]
```

Code Block 13 Levenshtein (Edit) Distance targetting GPU with floating point scores.

```
1 @vectron.gpu
2 def levenshtein(Q, T, gap, match, mismatch)
3   # Initialize M with dimensions (n+1) * (m+1) with zeroes
4   # Initialize the first row of M with row_i * gap
5   # Initialize the first column of M with column_j * gap
6   M = [[0.0] * (m + 1) for _ in range(n + 1)]
7   for i in range(n + 1):
8     M[i][0] = gap_o + i * gap_e
9   for j in range(m + 1):
10    M[0][j] = gap_o + j * gap_e
11
12  for i in range(1, n+1):
13    for j in range(1, m+1):
14      M[i][j] = max(
15        M[i - 1][j] + gap,
16        M[i][j - 1] + gap,
17        M[i-1][j-1] + (match if q[j-1] == t[i-1] else mismatch)
18      )
19  return M[n][m]
```