Using SIMD instructions to accelerate logic inside a RDBMS for bioinformatics sequence similarity searches*

Extended Abstract†

Sidath Randeni Kadupitige
Uwe Roehm
h.randeni@gmail.com
uwe.roehm@sydney.edu.au

ABSTRACT

CCS CONCEPTS

• **Information systems → DBMS engine architectures**;

KEYWORDS

Bioinformatics

ACM Reference format:
Sidath Randeni Kadupitige and Uwe Roehm. 2017. Using SIMD instructions to accelerate logic inside a RDBMS for bioinformatics sequence similarity searches. In *Proceedings of Insert conference Here, Unknown Location, June 2017 (UNKNOWN'17)*, 6 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

In this paper we explore using the new .NET Numerics.Vectors class library to implement accelerated bioinformatics applications inside an RDBMS. We do this by leveraging the SIMD instructions made available through the .NET Numerics.Vectors library and the new compiler in .NET 4.6. We use these new features to implement the SW algorithm in C# using a similar method to the Striped SW introduced by Farrar [4] and also implemented in [3] [15] [16]. Then we integrate the assembly into the RDBMS as a stored procedure.

2 BACKGROUND

2.1 Dynamic Programming

Dynamic programming is a technique used in many fields to solve complex problems by dividing the problem into several smaller problems and computing and storing the solutions to each of the smaller sub-problems. The solutions for the sub-problems are then combined and a solution to the initial complex problem is then found. In bioinformatics, it is used for sequence alignment. It's primary representation is in the Smith Waterman [13] and Needleman Wunsch [10] sequence alignment algorithms.

*Produces the permission block, and copyright information
†The full version of the author's guide is available as `acmart.pdf` document

UNKNOWN'17, June 2017, Unknown Location

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

	A	C	G	T	N
A	4	0	0	0	-2
C	0	9	-3	-1	-3
G	0	-3	6	-2	0
T	0	-1	-2	5	0
N	-2	-3	0	0	6

Table 1: BLOSUM62 matrix for our DNA alphabet of A, C, G, T, N

2.1.1 Smith Waterman. The Smith Waterman algorithm is a local alignment algorithm that finds the optimal local alignment between two sequences based on a given scoring matrix.
The algorithm itself rests on the use of a two dimensional scoring matrix H with one axis being the reference sequence d and one axis being the query sequence q. Each sequence string is pre-pended with a zero character which allows the first row and first column of the scoring matrix to be set to zero. Then each position in the matrix is calculated based on the following equation:

$$H_{i,j} = max \begin{Bmatrix} 0 \\ H_{i-1,j} - G_{init} \\ H_{i,j-1} - G_{init} \\ H_{i-1,j-1} + W(q_i, d_j) \end{Bmatrix}$$

Each horizontal or vertical movement computation is based on a gap score. Biologically speaking, it is more common that once an insertion or deletion has occurred, an extension of said insertion or deletion in more likely. This is shown in Figure [INSERT GRAPH SHOWING EXPECTED GAP PENALTY LINEAR GAP PENALTY AFFINE GAP PENALTY HERE]. There are two commonly used gap models:

(1) **Linear Gap Penalty**: Each consecutive gap of length k has a penalty score of $k * G_{init}$, where $k > 0$.
(2) **Affine Gap Penalty**: Each gap of length 1 has a penalty score of G_{init} and each consecutive gap of length $k > 0$ has a penalty score of $G_{init} + (k-1) * G_{ext}$

Each diagonal movement computation is based on a score profile W which contains all the characters found in the alphabet of q and d. A common set of score profiles are the BLOSUM DNA and protein profiles and the PFAM protein profiles [6]. We used a truncated BLOSUM62 scoring profile that only used the values for a DNA alphabet of **{A, C, G, T, N}** as shown in Table 1.

Once the scoring matrix has been fully computed, a traceback operation is carried out from the maximum score until a score of zero is reached. Figure 4 shows a completed score matrix H

and Figure 5 shows an example traceback operation on the same completed score matrix H. [INSERT FIGURE FOR FULL MATRIX, INSERT FIGURE FOR TRACEBACK ON FULL MATRIX]

2.1.2 Gotoh Extension. While the SW algorithm works fine for a linear gap penalty, some modification is required to compute for an affine gap penalty. These modifications were introduced by Gotoh [5] and involve adding two new scoring matrices E and F to keep track of vertical gaps and horizontal gaps. The score computation for $H_{i,j}$ is modified as follows:

$$H_{i,j} = max \begin{Bmatrix} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + W(q_i, d_j) \end{Bmatrix}$$

where $E_{i,j}$ and $F_{i,j}$ are computed as follows:

$$E_{i,j} = max \begin{Bmatrix} E_{i,j-1} - G_{ext} \\ H_{i,j-1} - G_{init} \end{Bmatrix}$$

$$F_{i,j} = max \begin{Bmatrix} F_{i-1,j} - G_{ext} \\ H_{i-1,j} - G_{init} \end{Bmatrix}$$

The values for $H_{i,j}$, $E_{i,j}$ and $F_{i,j}$ are equal to 0 when $i < 1$ or $j < 1$.

2.1.3 Needleman Wunsch differences. The first widely used dynamic programming algorithm for sequence alignment was the Needleman-Wunsch algorithm [10]. The NW algorithm looks for a global alignment between two sequences. As the SW algorithm is based on the NW algorithm, it uses similar principles. The key differences are as follows:

(1) **Initialization**: Only the origin position $H_{0,0}$ of the scoring matrix is set to 0. the remainder of the row can be computed.
(2) $H_{i,j}$ **calculation**: The requirement for a minimum score of zero is removed.
(3) **Traceback**: The starting position for the traceback is set to the final point of the scoring matrix $H_{|q|,|d|}$, not the maximal point. The termination condition for the traceback is now when both $i = j = 0$.

2.2 SIMD

In order to facilitate parallelism for common functions, CPU vendors began to offer certain instructions that allowed a single operation to be carried out on a prepared set of multiple data items. These instruction sets were bundled as extensions and tagged under the common headline - **Single Instruction Multiple Data** or **SIMD**. The most common extension sets in existing CPUs is SSE v4 with modern CPUs that were released after 2014 supporting a new set of extensions under the moniker AVX. The primary difference (apart from certain supported instructions) between these two extension sets is the size of the register upon which these extensions are executed. In SSE v4, the register size is 128bits, whereas with AVX, the register size is 256 bits. The register size determines how many data points can be operated on at once. E.g. if your data is based on 32bit ints, then a 128bit register would allow you to carry out operations on four 32bit ints, while a 256bit register would allow you to carry out operations on eight 32bit ints. Figure 1 shows a typical SIMD 128-bit and 256-bit register data partition. Figure 2

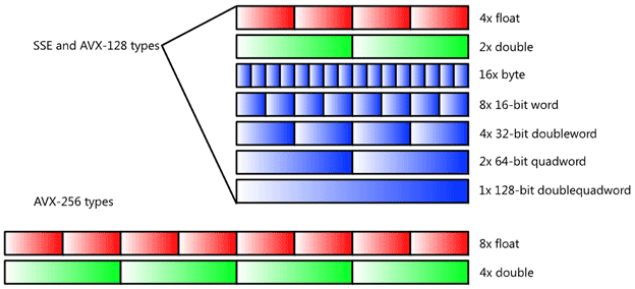

Figure 1: Data partitioning in a typical SIMD 128-bit and 256-bit register [9]

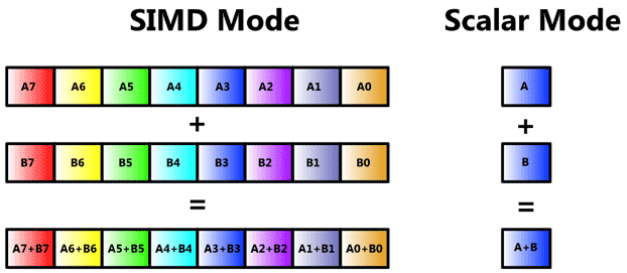

Figure 2: Example of a vectorised addition operation carried out on a SIMD register [9]

shows an example of a vectorised addition operation carried out on an SIMD register with a vector depth of 8.

From 2014, .NET 4.5 began testing a System.Numerics package that allowed for SIMD accelerated operations. In 2016 with the release of the new compiler, this package was officially supported as a part of .NET 4.6 [2]. As the package deals with managed objects as opposed to direct register operations, some operations such as shifting, shuffling, masks and horizontal max operations are not supported, but a sufficient portion of SSE4 and AVX operations are supported to allow for the development of a SIMD accelerated dynamic programming sequence alignment algorithm.

3 METHODOLOGY

3.1 SIMD accelerated Dynamic Programming

Since Wozniak Wozniak [14] first introduced the idea back in 1997, several different attempts to speed up the dynamic programming algorithms for sequence alignment have come into existence (outlined in Figure 3. All of them use the concept of a pre-computed query profile for matching every possible character in the target sequence alphabet. Currently the best intrasequence (comparing one sequence to the target sequence) is based on Farrar's method described in [4]. As our solution is based on this method, we will also compare to other SW aligners that use this method.

3.1.1 Farrar method. The Farrar method [4] for carrying out a SIMD accelerated striped Smith Waterman algorithm is the most

bp	max. score	bit-depth	vector type	elements sse	avx
15	135	8	byte	16	32
35	315	16	ushort	8	16
50	450	16	ushort	8	16
75	675	16	ushort	8	16
100	900	16	ushort	8	16
150	1350	16	ushort	8	16
300	2700	16	ushort	8	16
500	4500	16	ushort	8	16
1000	9000	16	ushort	8	16
1500	13500	16	ushort	8	16
3000	27000	16	ushort	8	16
5000	45000	16	ushort	8	16
10000	90000	32	uint	4	8
20000	180000	32	uint	4	8
40000	360000	32	uint	4	8
$1x10^5$	$9x10^5$	32	uint	4	8
$1x10^6$	$9x10^6$	32	uint	4	8
$1x10^7$	$9x10^7$	32	uint	4	8
$1x10^8$	$9x10^8$	32	uint	4	8
$1x10^9$	$9x10^9$	64	long	2	4
$3x10^9$	$27x10^9$	64	long	2	4

Table 2: SIMD query profile assigned bit-depths, vector types and vector elements

efficient for *intrasequence* sequence alignment (some better methods exist for *intersequence* sequence alignment [11]) and is it integrated into some common short read aligners such as BWA-SW [8], bowtie2 [7] and novoalign [1]. The key innovation of Farrar's approach over previous methods such as Wozniak [14] and Rognes [12] was the use of a striped query profile. Figure 3 from [11] outlines the various different query profiles that were considered for SIMD-accelerated SW. Farrar's approach is shown in example C of Figure 3. By compiling SIMD vectors from every t-th character in the query, Farrar was able to uncouple the dependencies within the two nested loops for iterating across the score matrix. A query profile is a set of pre-computed alignment scores of the query against the alphabet of the target sequence. Usually the alphabet for both the query and the target sequence are the same. In our (and most other) implementation, our expected alphabet is {**A**, **C**, **G**, **T**, **N**, -} for DNA sequences. So, our query profile would take the form of the reference alphabet size of 6 times t vectors of size determined by the query size shown in Table 2. The psuedocode for the Farrar method is outlined in Algorithm 1 for query profile creation and Algorithm 2 for score matrix calculation.

3.2 Our Implmentation

Our implementation follows Farrar's approach, but has a few differences due to the limitations imposed by the System.Numerics.Vector library in .NET 4.6. Our approach is outlined in Algorithm 3.

Algorithm 1 Query Profile creation

```
function CreateQueryProfile(Query, Reference, bitDepth)
    segLen ← (Query.length + bitDepth − 1)/bitDepth          ▷ integer value set
    for all a in Reference.Alphabet do
        h ← 0
        for i = 0..segLen do
            j ← i
            for k = 1..bitDepth do
                if j > Query.length then
                    queryProfile[a][h] ← 0
                else
                    queryProfile[a][h] ← W(a, q[j])
                end if
                h ← h + 1
                j ← j + segLen
            end for
        end for
    end for
    return queryProfile
end function
```

Algorithm 2 Score Matrix Calculation

```
function TraverseScoreMatrix(Query, Reference, vQueryProfile, segLen)
    for i = 0..Reference.length do
        vF ←< 0, ..., 0 >                                    ▷ Zero Vector
        vH ← vHStore[segLen − 1] << 1
        Swap(vHLoad, vHstore)
        for j = 0..segLen do
            vH ← vH + vQueryProfile[i][j]
            vMax ← max(vMax, vH)
            vH ← max(vH, vE[j])
            vH ← max(vH, vF)
            vHStore[j] ← vH
            vH ← vH − vGapOpen
            vE[j] ← vE[j] − vGapExtend
            vE[j] ← max(vE[j], vH)
            vF ← vF − vGapExtend
            vF ← max(vF, vH)
            vH ← vHLoad[j]
        end for
        vF ← vF << 1
        j ← 0
        while AnyElement(vF > vHStore[j] − vGapOpen) do
            vHStore[j] ← max(vHStore[j], vF)
            vF ← vF − vGapExtend
            if + + j >= segLen then
                vF ← vF << 1
                j ← 0
            end if
        end while
    end for
    return vHStore
end function
```

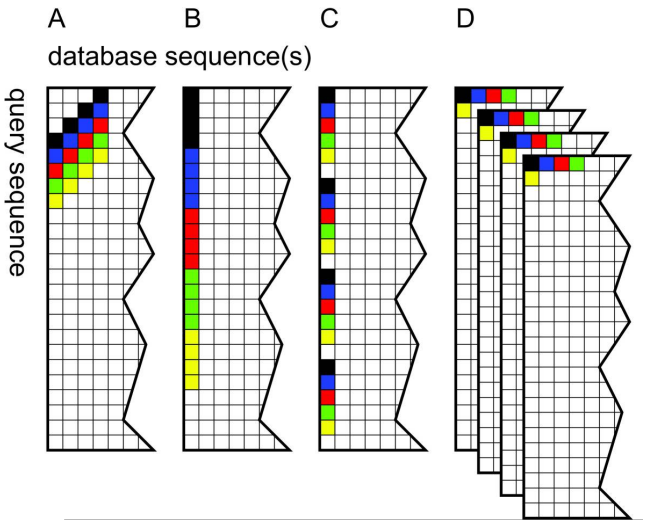

Figure 3: Examples of various query profiles. A: Vectors along a diagonal [14]; B: Vectors along the query [12]; C: Striped Vectors along the query [4]; D: Vector across multiple queries [11]

3.3 RDBMS Integration

Given we carried out our implementation in C# (which runs on the CLR) and our RDBMS also runs on the CLR, porting our solution into a stored procedure was trivial.
DOES THIS NEED TO BE MADE MORE PROPER?
We took the console programs primary called function from class **Progam** - **Main(string[] args)** and moved it into a stored procedures class with a new name **SWAlignment()**. Then we use our IDE (MS Visual Studio 2017) to build and publish the stored procedure to our RDBMS. Once this is done, we can call it using the MSSQL command **exec AlignmentTestDBSP1**

4 EVALUATION

Table 3 and Table 4 show an average runtime for various query sizes and edit distances for the generic Smith Waterman algorithm and the SIMD improved version. In order to get a more fine-grained reading for performance at this level, we used elapsed cpu ticks instead of milliseconds as our counter. The frequency for these runs was: [INSERT CPU FREQUENCY HERE]

When we chart the values in log scale as seen in Figure 4 and Figure 5, we can see that the SIMD improvements are resulting in an order of magnitude improvement at the higher query lengths. Since read sizes are continually increasing, this does bode well for the performance of our solution when measured against upcoming read sizes.

TBD Need to add comparisons to SWIPE, Parasail, SSW, libssa, opal.
TBD Need to find a way to convert runtime comparisons from various

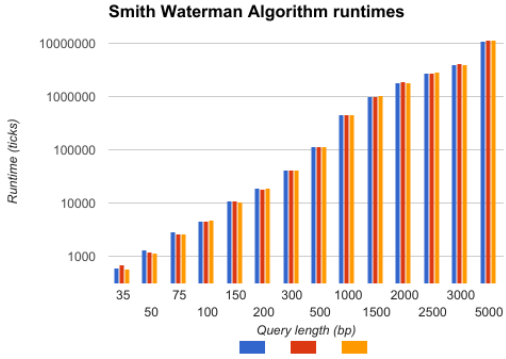

Figure 4: Smith Waterman Algorithm runtimes without any SIMD acceleration.

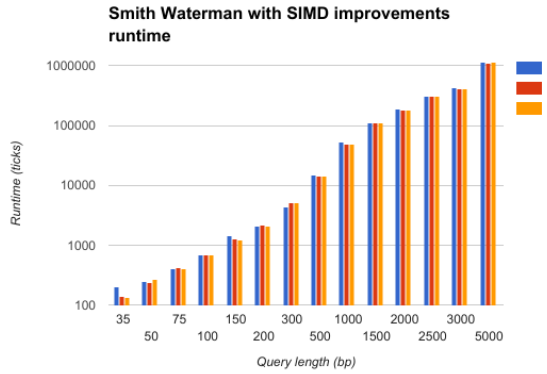

Figure 5: Smith Waterman Algorithm runtimes with SIMD acceleration.

Size (bp)	ed 0	ed 5	ed 10
35	605.6	678	580.8
50	1316.2	1216	1131.4
75	2892.6	2671.8	2582.6
100	4621	4648.6	4874.8
150	10736.2	10805.6	10392.4
200	18824.2	18527.2	18857.8
300	41141.6	41437.2	41468.6
500	113543	114351.6	113188
1000	447720.6	450967.2	452378
1500	1010722.2	1007057.4	1053102
2000	1789257.2	1869152.2	1824046.6
2500	2783786.6	2814603.6	2861773
3000	4034815.8	4134108.4	4040465.8
5000	11191762.2	11373476.2	11378558

Table 3: Runtimes for running smith waterman algorithm with linear gap penalties

Algorithm 3 Our Score Matrix Calculation Implementation

```
function TraverseScoreMatrixNew(Query, Reference, vQueryProfile, segLen, gapPenalty)
    vZero ←< 0, ..., 0 >                                  ▷ Zero Vector
    vGap ←< gapPenalty, ..., gapPenalty >                 ▷ GapCost Vector
    for i = 0..Reference.length do
        for j = 0..segLen do
            if i == 0 then
                vH ← max(vZero, vQueryProfile[j][Reference[i]])
            else
                if j == 0 then
                    vH ← matrixProfile[segLen − 1][i − 1]
                    vH ← vH >> 1
                    vH ← max(vZero, vH + vQueryProfile[j][Reference[i]])
                else
                    vH ← max(vZero, matrixProfile[j − 1][i − 1] + vQueryProfile[j][Reference[i]])
                end if
                vE ← matrixProfile[j][i − 1] − vGap
                vH ← max(vH, vE)
            end if
            matrixProfile[j][i] ← vH
        end for
        changeFlag ← false
        for j = 0..segLen do
            if j == 0 then
                vH ← matrixProfile[segLen − 1][i]
                vH ← vH >> 1
                vF ← vH − vGap
            else
                vF ← matrixProfile[j − 1][i] − vGap
            end if
            greaterThanFlag ← greaterThanAny(vF, matrixProfile[j][i])
            changeFlag ← changeFlag||greaterThanFlag
            vH ← max(vF, matrixProfile[j][i])
            matrixProfile[j][i] ← vH
            if (j == (segLen − 1)) ∧ changeFlag then
                j ← −1
                changeFlag ← false
            end if
        end for
    end for
    return matrixProfile
end function
```

5 CONCLUSION

While we are an order of magnitude slower than the C or C++ implementations, we do see an order of magnitude increase compared to a non-SIMD accelerated version in C#.

6 FUTURE WORK

GPU accelerations inside the RDBMS.

Size (bp)	ed 0	ed 5	ed 10
35	202.4	141	133.6
50	247.2	238	262.6
75	402.4	412.6	396.2
100	682.2	684.8	685
150	1423.6	1296.8	1244.4
200	2076.8	2147.2	2093.4
300	4409.8	5157.6	5215.4
500	14792	14250.4	14353
1000	52660.6	49817.8	49860.4
1500	110102	112048.8	112957.2
2000	185510.2	179349.2	182695.6
2500	308050.2	307851	307417
3000	420564.6	419879.4	407331.2
5000	1131352.4	1112231.8	1132206.2

Table 4: Runtimes for running smith waterman algorithm with linear gap penalties using SIMD accelerated code

Size (bp)	ed 0	ed 5	ed 10
35	762.6	893	873.2
50	1665	1981.8	1697.4
75	3598.4	3819	3486.2
100	6228.8	7587.6	6778.2
150	13775.2	20191.4	17887.2
200	22734.6	27976.6	26201.2
300	51569.2	54525.4	57098.4
500	140972.2	164566.2	151273.2
1000	573196	600408.6	656190.4
1500	1255819.6	1281211	1414785.8
2000	2209770.8	2956804.6	2401326.2
2500	3533241.8	3797596	3823031.2
3000	5098129.6	5489876.2	5493088
5000	16221085.2	16232273.6	15372186.4

Table 5: Runtimes for running smith waterman algorithm with linear gap penalties inside SQL Server CLR

ACKNOWLEDGMENTS

REFERENCES

[1] Novocraft Technologies Sdn Bhd. 2014. NovoAlign. (2014). http://www.novocraft.com/products/novoalign/

[2] Microsoft Corporation. 2017. .NET Framework 4.7, 4.6, and 4.5. (2017). https://msdn.microsoft.com/en-us/library/w0x726c2(v=vs.110).aspx

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

[4] M. Farrar. 2006. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* 23, 2 (nov 2006), 156–161. https://doi.org/10.1093/bioinformatics/btl582

[5] O. Gotoh. 1982. An improved algorithm for matching biological sequences. *J. Mol. Biol.* 162, 3 (Dec 1982), 705–708.

[6] S. Henikoff and J. G. Henikoff. 1992. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences* 89, 22 (nov 1992), 10915–10919. https://doi.org/10.1073/pnas.89.22.10915

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

[8] H. Li and R. Durbin. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* 25, 14 (may 2009), 1754–1760. https://doi.org/

Size (bp)	ed 0	ed 5	ed 10
35	294	169.8	322
50	414.2	242	268.4
75	431.2	398.8	383.6
100	654.4	575.2	595.2
150	1142.4	950.2	1078
200	1889.8	1675.6	2013
300	10221.2	8961	7712.8
500	15345.2	21686	17198.2
1000	44999.2	41521.2	64469.6
1500	89420.6	91059.4	83718
2000	147050.4	154474.4	150111.2
2500	217245	225807.6	235809.4
3000	310610	319337	350907.6
5000	903959.8	892981.2	1053578.2

Table 6: Runtimes for running smith waterman algorithm with linear gap penalties using SIMD accelerated code inside SQL Server CLR

10.1093/bioinformatics/btp324

[9] Chris Lomont. 2011. Introduction to Intel® Advanced Vector Extensions. (2011). https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions

[10] 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 (mar 1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

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

[12] T. Rognes and E. Seeberg. 2000. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics* 16, 8 (Aug 2000), 699–706.

[13] T. F. Smith and M. S. Waterman. 1981. Identification of common molecular subsequences. *J. Mol. Biol.* 147, 1 (Mar 1981), 195–197.

[14] A. Wozniak. 1997. Using video-oriented instructions to speed up sequence comparison. *Comput. Appl. Biosci.* 13, 2 (Apr 1997), 145–150.

[15] Mengyao Zhao, Wan-Ping Lee, Erik P. Garrison, and Gabor T. Marth. 2013. SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications. *PLOS ONE* 8, 12 (12 2013). https://doi.org/10.1371/journal.pone.0082138

[16] Martin ÅăoÅąic. 2015. *An simd dynamic programming C/C++ library.* Master's thesis. University of Zagreb. https://bib.irb.hr/datoteka/758607.diplomski_Martin_Sosic.pdf