Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Thesis

Design and FPGA implementation of a protein structure comparison method based on alignment of backbone conformations

Árpád Goretity
Molecular Bionics Engineering BSc

2016

Advisors:
Zoltán Nagy, PhD
Zoltán Gáspári, PhD

Table of Contents

Kivonat

A fehérjék az élő rendszerek legfontosabb építőelemei közé tartoznak. A XX. és XXI. század számítástechnikai-informatikai forradalmának köszönhetően óriási fizikai, kémiai és információelméleti komplexitásuk végre megadni látszik magát a különböző elemző-, összehasonlító, és egyéb, a kutatást segítő bioinformatikai módszereknek és algoritmusoknak. A fehérjék átfogó számítógépes vizsgálata előrehaladást jelenthet az evolúció pontosabb és jobb megismerésében, s ezáltal — közvetlenül vagy közvetetten — például genetikai betegségek felismerésében, megelőzésében, gyógyításában is.

A térszerkezeti illesztő- és összehasonlító metódusok különösen alkalmasak a fehérjék közötti különféle összefüggések széleskörű vizsgálatára. Sajnálatos módon a jelenleg használatban lévő algoritmusok oly módon teszik fel a hasonlóság mértékére és mibenlétére vonatkozó kérdéseket, hogy azoknak a megválaszolása NP-nehéz, így erőteljesen számítás-, teljesítmény- és időigényes. Alkalmanként az irodalomban is megemlítik, hogy ez nehézkessé, sőt, olykor egyenesen lehetetlenné teszi a közepes vagy nagyobb adatbázisokkal való munkát, az azokban szereplő fehérjék páronkénti, kimerítő összehasonlítását.

A magas komplexitásosztályokba tartozó algoritmusok gyorsítására bevett módszer a heurisztikák alkalmazása. Ezeknek azonban megvan az a hátulütője, hogy információvesztéssel járhatnak — természetükből fakadóan előnyben részesíthetnek például egy könnyebben, gyorsabban elérhető lokális optimumot az elvárt globális optimummal szemben. Általában a kettő közötti különbség kicsi, és nem vezet nagy hibához, azonban léteznek olyan szélsőséges esetek, amelyekben az elvesztett globális optimum jelentős problémát okozhat.

Jelen dolgozat tárgya egy újfajta térszerkezeti fehérjeillesztő módszer kidolgozása, amely a Smith–Waterman-szekvenciaillesztőn alapul. Mint dinamikus programozási algoritmus, a Smith–Waterman-eljárás mindig megtalálja a matematikai értelemben vett legjobb megoldást, emellett futásideje is mindössze négyzetes. Tervünket implementáljuk egy programozható logikai áramkörön, nevezetesen FPGA- (Field-Programmable Gate Array) alapú FPSoC- (Field-Programmable System-on-a-Chip) rendszeren, ami a masszív párhuzamosítás általi hardveres gyorsításra is lehetőséget nyújt. Az újonnan elkészített hardvert összehasonlítjuk eddigi, azonos célú módszerekkel — számítási képesség, sebesség, skálázhatóság, energiahatékonyság illetve hasznos hardverkihasználtság terén. A dolgozatban megvalósított algoritmus egy kezdeti változatával már vonatkozó munkákban megismerkedhettünk; céljaink között az is szerepel, hogy azt mind memóriaigény, mind futási sebesség szempontjából továbbfejlesszük és optimalizáljuk.

Abstract

Proteins are one of the most important building blocks of biological systems. Their tremendous physical, chemical and information-theoretical complexity finally seems to succumb to a wide variety of analytical and comparative research methods and bioinformatic algorithms, thanks to the progress in information technology and computing during the 20th and 21st centuries. Comprehensive analysis of proteins may potentially lead to more precise knowledge of evolution, thereby directly or indirectly facilitating the identification, prevention and treatment of genetic disorders.

Structure alignment and structural comparison methods in particular allow us to explore different aspects of relationships between proteins. Unfortunately, state of the art algorithms tend to formulate the question of similarity in such a way that the solution is NP-hard, therefore highly computationally intensive, resource- and time-consuming. As it is noted on some occasions in the literature, this makes it very hard, or even completely impossible, to carry out exhaustive pairwise comparison between every protein in a medium-sized or large database.

It is common practice to attempt to speed up these complex programs using heuristics. This, however, results in loss of information, since such heuristics might miss the globally optimal solution if a local optimum is significantly easier or faster to find. Most of the time, the difference is small and a local optimum is an acceptable final outcome, but in certain edge cases, this mistake may still be significant.

In this thesis, we present the design of a novel structural alignment method based on the Smith–Waterman sequence alignment algorithm. Being a dynamic programming approach, Smith–Waterman is guaranteed to find the globally optimal alignment, while running in only quadratic time. We implement the algorithm on a Field-Programmable System-on-a-Chip computing platform based on a Field-Programmable Gate Array that enables hardware acceleration through heavy parallelization. We compare the new method to existing structural alignment techniques with regards to computational complexity, performance, scalability, energy efficiency, and we also assess its effective hardware use. An initial variation of this procedure is rooted in previous, related work; our goal is to optimize its implementation further in terms of memory consumption and physical execution time.

1. Introduction

Modern biology is seeking answers to fundamental questions at the molecular, biophysical and biochemical level. The studying of such low-level internal structure of living matter is greatly aided by computational techniques. The ability to execute millions and billions of elementary mathematical operations in a second opens up opportunities that were previously unheard of. Many forms of simulation and numerical analysis make it possible for researchers to examine real or hypothetical phenomena that would otherwise be very hard, prohibitively expensive or even impossible to realize in a wet lab, and to process high amounts of data that could not be directly understood by humans in reasonable time.

In the center of attention are proteins, an outstandingly important group of biological compounds. They are responsible for many features of organisms, which is possible due to their high complexity. Despite being built from only 20 to 23 types of so-called proteinogenic amino acids (the exact number depending on their classification), their relatively long nature gives rise to great variance. For example, a protein chain of length 300 (which is only an average length [1] [2]) can be assembled in $V_{300}^{20} = 20^{300}$, or approximately $2.04 \cdot 10^{390}$ different ways. As well as enabling proteins to have a multitude of functions, this complexity makes the life of the scientist harder. Therefore, a corner stone of current bioinformatics research is finding new, more efficient ways of dealing with biological-scale complexity in proteins and other compounds, in order to be able to learn more about their role in life and evolution.

The intent of this study is to improve upon the effectiveness and resource usage of existing solutions, by means of combining mathematical models and algorithms in new ways, and integrating them with high-performance computational techniques. We first give a brief outline of the current state of the field and define the scope of our work, then elaborate on the design details of the system. Finally, we analyze the gained advancements and appoint the direction of future research.

1.1. Classification of Methods for Structural Analysis

Answering different kinds of questions relevant to evolutionary, chemical, physical, and functional features of proteins requires different analytical methods. In the following section, we lay out the most important definitions that permit us to be precise about their goals, implementation, advantages, disadvantages, and whether our work is of significant relevance to each of them.

1.1.1. Structure Superposition

Structure superposition is the act of searching for the best match between two sets of corresponding points in three-dimensional space, for some — mathematical — definition of “best.” Points usually represent atoms of a molecule. By nature, superposition problems always have an exact, globally optimal solution. The key observation to be made is that the correspondence of the points is a known piece of information, and that is what makes this task simpler than most other structural analysis problems. In fact, in the easiest case, the very sequences of the two structures are identical (for instance, when two conformers of the same compound are considered), so the correspondence is trivial. This means that the optimal match can be obtained via simple least-squares fitting.

1.1.2. Structure Alignment

The aim of structure alignment is finding the best correspondence between three-dimensional points in two different sets. This means that the mapping between the points is not known in advance. As a consequence, this problem is a lot more difficult to solve than superposition; in the general case, it is NP-hard. Apart from having great asymptotic complexity, solving the problem has additional practical issues. For example, two similar proteins may have individual, corresponding structural elements in a different order within the sequence, which means that a naive, “linear” approach might only be able to fit the two molecules partially. Hence, permuting the appropriate subsequences may be needed. For this to be computationally feasible, some heuristics should be devised for finding the boundaries of each bigger substructure: trying every possible permutation of each amino acid in a protein would take too long, as the number of permutations grows superexponentially with the length of the chain.

1.1.3. Structure Comparison

Structure comparison seeks to describe the degree of similarity between two structures. Results may be obtained by alignment or by some other means. Although aligning the subject sequences or structures might seem an obvious first take, it is not necessarily the best solution, especially when structural similarity is inferred from the similarity of sequences. A structural comparison algorithm without alignment is PRIDE [3], and its amended variant, PRIDE2 [4]. The basis of these is the distribution of intermolecular distances. In particular, $C\alpha_i - C\alpha_{i+n}$ distances are analyzed in both structures for $n = 3 \ldots 30$. The resulting pairs of distributions are then correlated and analyzed using contingency tables, then the similarity score is derived from an overall average probability measure.

All of the methods described above can be used for local and global search. In our work, we focus on local alignment and comparison, therefore we will not discuss or try to improve upon global alignment methods or on those suited for structure prediction.

1.2. Theoretical Basis of Structural Comparison of Proteins

The function of individual proteins is sometimes defined by, but at least strongly correlated with, their spatial structure. Hence structural analysis is one of the primary methods used in the study of proteins. In particular, comparison of two (or more) such structures allows one to relate the corresponding proteins with regards to their evolutionary relationship, quantitatively assess their similarity, and predict their function and the expected biological effects of certain modifications on them, making structural comparison a valuable member of the biomedical researcher's toolbox.

Unfortunately, existing approaches to structural comparison and prediction are computationally intensive. Specifically, current formulation of structural alignment and threading (protein fold recognition) problems tends to be such that finding a globally optimal solution to them is NP-complete or NP-hard [5].

1.3. Issues of Current Structural Alignment Methods

1.3.1. Methods Based Partly or Entirely on Amino Acid Sequences

Structural prediction software that tries to compute structure using the amino acid sequence of a protein exists since the late 90's. Most of the applications in this class rely on databases of experimental results and the fact that in practice, there is a relatively small number of substantially different folds, so querying a database and calculating a similarity score in order to find partly or perfectly matching results is feasible.

Such a collection is CATH [6] [7], the Class-Architecture-Topology-Homology database, which classifies proteins and domains at four levels, based on their structural information of different granularities. Class-based (C) grouping operates based on the qualitative assessment of secondary structure (whether the domain is an alpha helix, a beta sheet, a mixture of these or none of them). Architectural (A) classification respects actual three-dimensional structure and placement of the domains, while topological (T) classification is based on the protein fold and the connections between secondary structure elements. Finally, at the homologous superfamily (H) level, proteins are assigned the same superfamily if there exists provable evolutionary relationship between them.

Structural comparison based on this database and similar solutions can be quite accurate within some limits, but there are caveats that might reduce their reliability. Specifically, CATH is created by manual curation and a pipeline of various matching, searching and scoring programs. Although the authors of this database do use structure comparison techniques (for example, SSAP [8] [9]), the first step in the classification process for a new protein is not based on structure. When adding a new protein, a program called `cath-resolve-hits` is invoked first. The purpose of this tool is to reduce a list of potentially redundant domain matches to a set of optimal, non-overlapping subsequences, for which it uses amino acid sequence alignment. It then searches the database, and if it discovers similarity sufficiently close to the sequences of already-existing entries, it adds the new protein to the same class in which the aforementioned matches were found.

This heuristic works well most of the time, but there are exceptions. While there is definite correlation between the amino acid sequence and the spatial structure of a protein, their correspondence is much more complicated than what could simply and reliably be inferred algorithmically. The main issue is that the actual fold of a protein or that of a subsequence depends on its local environment, that is, exactly which surrounding molecules it forms coordination complexes with. This means that strictly speaking, even a perfect alignment of identical subsequences does not necessarily imply identical structures [10]. This is a real problem which is infrequent but not impossible to be encountered by those working with CATH.

1.3.2. Methods Based on Relative Positions and Distances

A more direct approach to structural comparison is the use of actual geometric data, acquired experimentally. The advantage of this family of algorithms is that the potential single point of failure of mispredicting physical structure based on sequences or that of incorrectly assuming correlation between the two is not present.

One of the most popular position-based alignment algorithms is DALI [11]. This represents the structure of an individual protein as a matrix D, where each row and column is associated an amino acid in the sequence. Every element D_{ij} of this matrix is then assigned a value proportional to the distance between the two amino acids corresponding to the row (i) and column (j) index of the element. As a result, the structure is recorded in a format that is independent of the position of the origin of the coordinate system in which the key points of the structure are placed. A slight problem with this representation is that it is redundant: since distance is an unsigned quantity, it is necessarily the case that $D_{ij} = D_{ji}$, therefore approximately twice as much memory is used as would strictly be necessary.

In order to perform the actual alignment, DALI breaks the global distance matrices into smaller (typically 6×6) overlapping submatrices — these correspond to contact patterns between hexapeptide groups within the same protein. The algorithm then searches for similar contact patterns in the two matrices, and chains pairs of them together if they share an already-associated pattern. The chain of maximal length corresponds to the best alignment.

Another well-known method is CMO, the Contact-Map Overlap Maximization algorithm [12]. This approach reformulates the question of structural similarity search as a graph-theoretic problem. For both proteins to be compared, it constructs a graph, the so-called contact graph, from the local environment of each C_α-atom by considering the Euclidean distances to some of its closest neighbors. It then searches for similarity by trying to map the vertices of one graph to those of the other, and maximizes the number of matches with the constraint that for many mapped vertices, the pattern of neighborhoods be similar.

DAST, the Distance-based Alignment Search Tool [13] improves upon CMO. One problem with CMO is that it may return results with high root mean square deviations.

This quantity is a measure of dissimilarity of two aligned proteins, and it may be defined in two different ways:

$$RMSD = \sqrt{\sum_{i=1}^{n} d(P_i^1, P_i^2)}$$

or

$$RMSD = \sqrt{\frac{1}{n}\sum_{i=1}^{n} d(P_i^1, P_i^2)}$$

where P^1 and P^2 are the aligned or mapped parts of the two protein sequences, and $d(x, y)$ is the Euclidean distance of the corresponding residues x and y. The reason for this is that in trying to maximize the number of aligned residues, it may associate physically distant atoms. DAST attempts to correct this behavior by — intuitively speaking — only considering two respective pairs of residues from the two proteins as a match when the absolute difference of the distances between the members of each pair is below a threshold τ. Using the notation of the DAST authors, the unordered pair of residues (i, j) in one protein may be aligned with another pair (k, l) in the other one if $|d_{i,j} - d_{k,l}| \leq \tau$. This means that — depending on the definition — the RMSD is either bounded by a constant factor or it is linear in the square root of the number of aligned residues.

All of the methods presented above solve problems that are generally NP-hard or NP-complete. The authors of [12] present a polynomial-time solution to a restricted subset of the problem, but even this means a complexity of $\mathcal{O}(n^3 \log n)$; furthermore, they mention that in previous work they have built upon, "for computational reasons, only small protein structures were tried." [14] Given that high time complexity only permitted verification of the latter methods using a small number of short protein chains, it might be the case that the results are biased in some form or another, i.e. that the results represent only those proteins that were used in the verification step, and that they might not be as biologically relevant as necessary. Meanwhile, DALI has seen improvements in efficiency as well. The authors of DaliLite v3 [15] apply search space pruning, by prefiltering the input based on prior knowledge about its distribution. Their claim is that this performance boost is required merely to keep up with the exponentially growing number of known structures.

These facts make it clear that we need to turn to new methods if we are willing to escape the realm of NP-hardness and NP-completeness. In the next chapter, we lay out the theoretical foundations for such a novel algorithm that combines the efficiency and optimality of existing fast dynamic programming techniques with the effectiveness of spatial structure comparison.

2. In Search of a New Method

2.1. Comparison of Protein Structures using Dihedral Angles

One of the most characteristic components of the spatial structure of proteins is their backbone conformation. This is described by two torsion or dihedral angles around every nonterminal alpha carbon: Φ, which is facing the neighboring nitrogen atom, and Ψ, which is directed towards the carbon atom of the next bond. Due to the high rigidity of the two amide planes vicinal to the alpha carbon (figure 2.1), it is almost only these two angles out of many possible degrees of freedom that affect the structure significantly.

In many proteins, conformation determines or at least influences function through various steric effects. Therefore, proteins that feature similar dihedral angles around amino acids at the same positions (indices) may exhibit analogous biochemical behavior, regardless of specific chemical composition (i.e., identity of the amino acids in question). Structural similarity might also be a sign of evolutionary connection that would otherwise remain undiscovered because of a point mutation, for example. Consequently, we can expect that structural comparison methods, in particular those based on backbone conformations, will yield different, and in some cases better, results than those that operate on amino acid sequences.

Due to steric and energetic reasons, protein backbones cannot have arbitrary conformations. By plotting physically advantageous, neutral but allowed, and prohibited (i.e., rarely or never-occurring) Ψ angles against the corresponding Φ angles in a Cartesian coordinate system, one obtains the so-called Ramachandran plot (depicted in figure 2.2), named after Indian biophysicist G. N. Ramachandran who first described it. It is also possible to plot the energy as a function of torsion angles, $E = f(\Phi, \Psi)$. This leads to a Ramachandran surface (figure 2.3), which can also be used for predicting preferred conformations.

One can define a metric between the points of the Ramachandran plot as a plane. This will, in turn, be useful both as a heuristic and as a scoring function for the quantitative analysis of structural similarity. In our work, we opted to use the negative square of the Euclidean distance (P=2 norm) added to an arbitrary constant, because implementing this formula is easy and fast on modern hardware, and it is also closely related to physical distance between atoms when the structures being examined are similar.

The reason why the difference of angles is related to the spatial distance between atoms of two similar structures is the following. When two structures are alike, we expect the differences between pairs of their dihedral angles to be small.
For two points P and P', both of which are at a distance r from the origin O, it is true

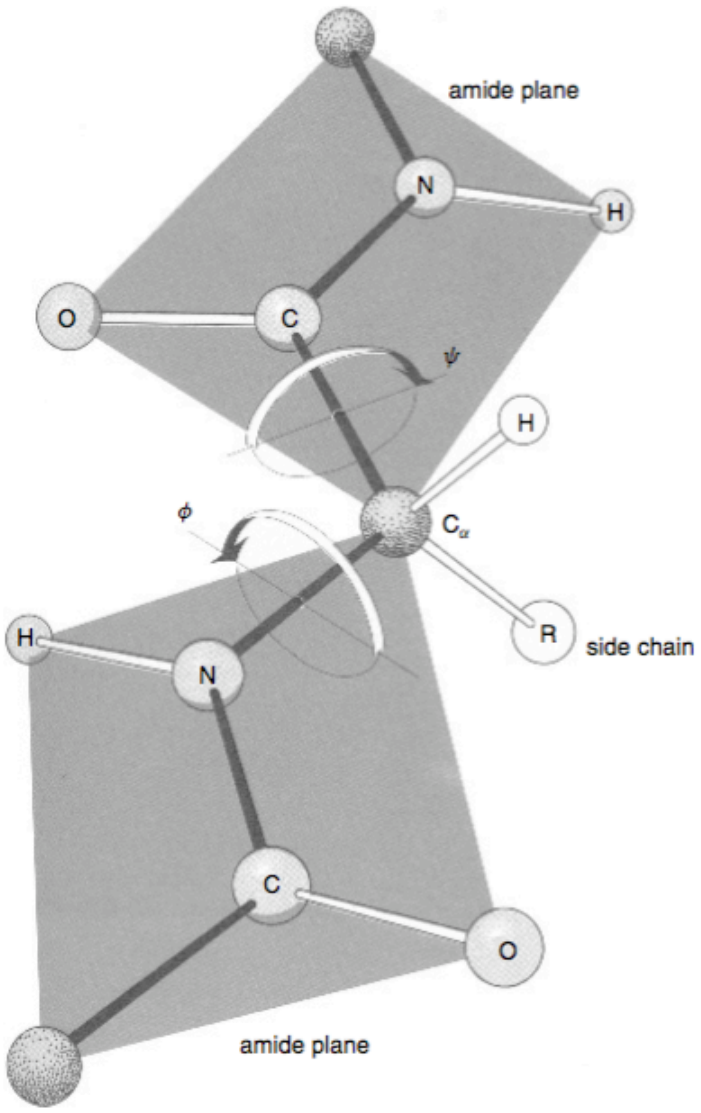

Figure 2.1. Peptide bonds and dihedral angles, Φ and Ψ. From P. E. Bourne and H. Weissig, *Structural bioinformatics*. Wiley-Liss, Inc., 2003.

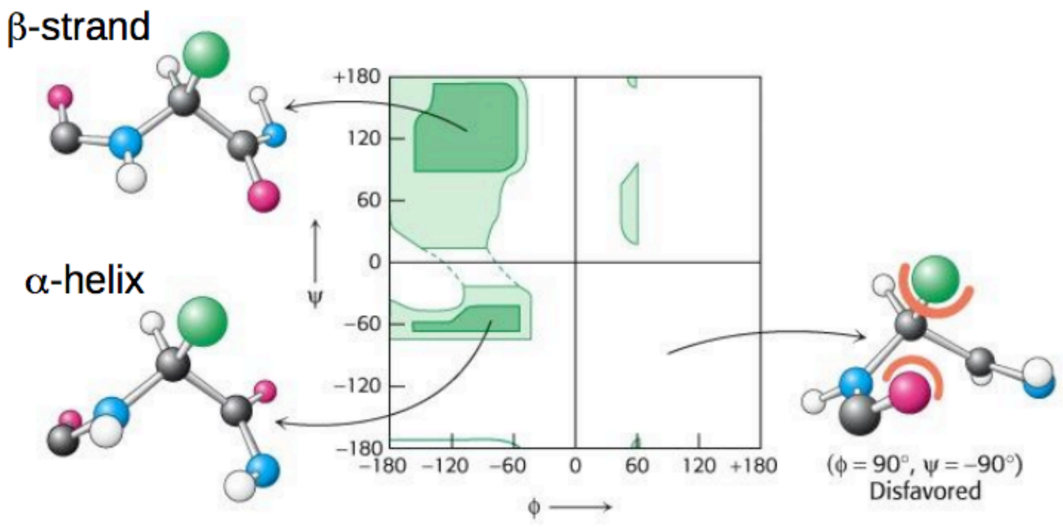

Figure 2.2. The Ramachandran plot. Preferred, allowed and forbidden regions are marked with dark green, light green and white color, respectively. From Z. Gáspári, “Structure validation”, University Lecture, 2016, [Online]. Available: `https://wiki.itk.ppke.hu/twiki/pub/PPKE/StructBioinf/StructureValidation.pdf`.

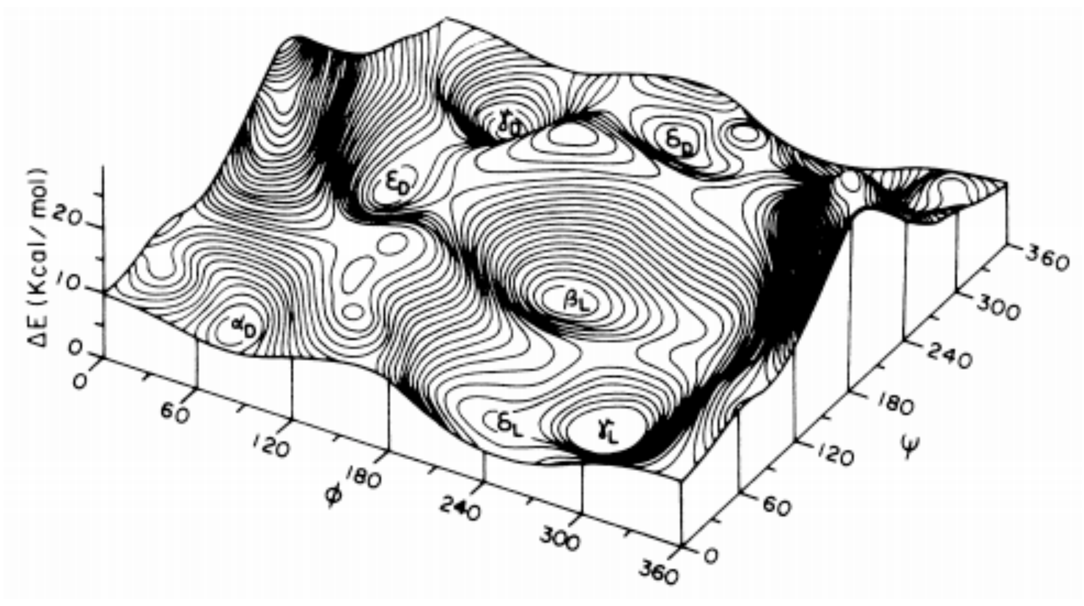

Figure 2.3. The Ramachandran surface. This depiction of energy levels is customary in computational chemistry. From Z. Gáspári, “Structure validation”, University Lecture, 2016, [Online]. Available: `https://wiki.itk.ppke.hu/twiki/pub/PPKE/StructBioinf/StructureValidation.pdf`.

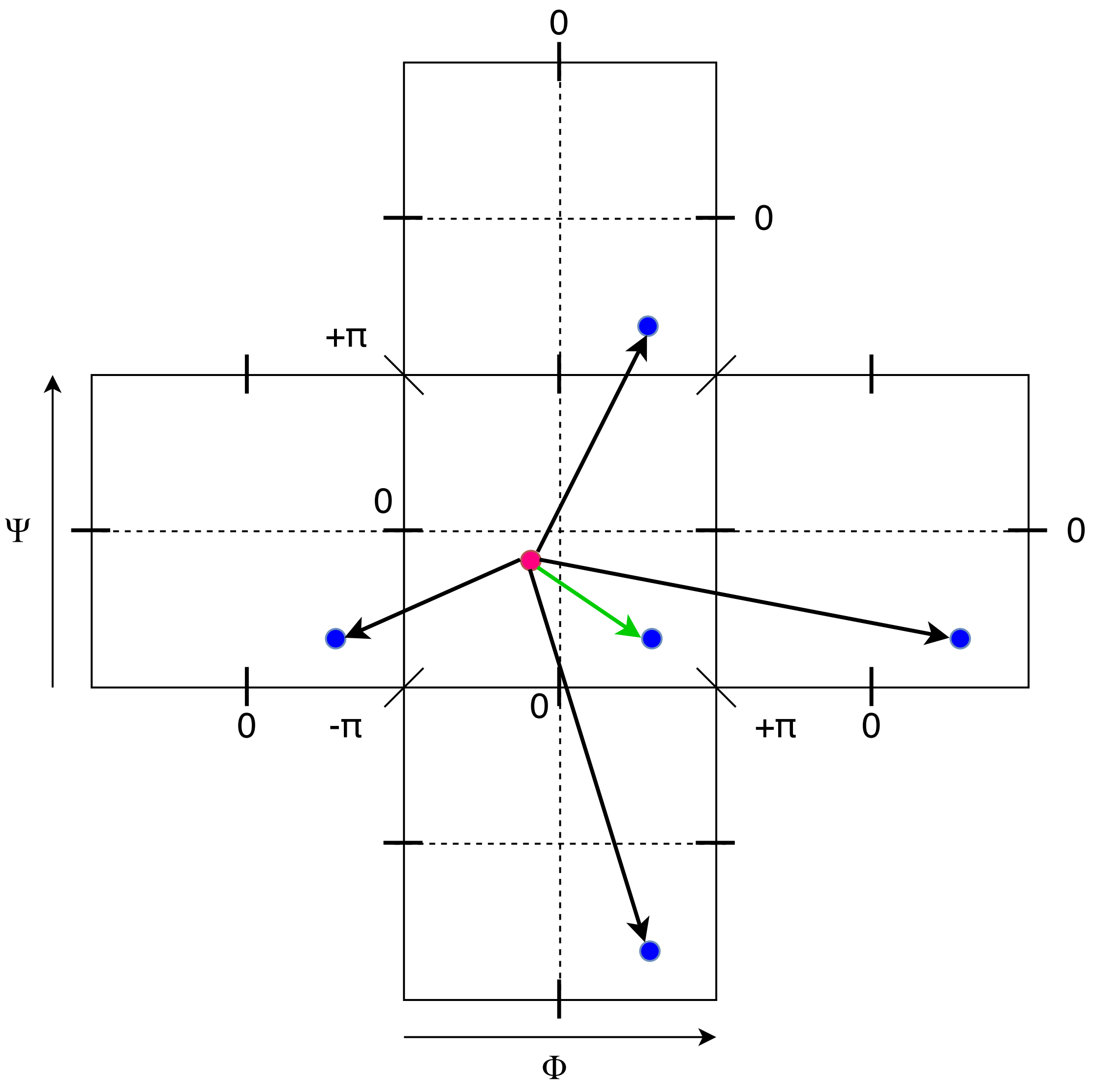

Figure 2.4. Periodicity of dihedral angles. The red dot marks one angle pair, while the blue dots show possible relative positions of another angle pair. The green arrow designates the shortest distance between the angle pairs.

that their distance $d(P, P') \approx r \cdot \sphericalangle POP'$, when $\sphericalangle POP'$ is sufficiently small. Therefore, we felt that the Euclidean distance squared in (Φ, Ψ) space would be a reasonable initial guess for the scoring heuristic.

When working with dihedral angles, one has to keep one caveat in mind. Torsion angles are angles of rotation, thus they are periodic with a period of 2π ($360°$). For example, an angle of $60°$ means exactly the same physical arrangement as $420°$. In addition, computing the difference of two angles is not as simple as a subtraction. The actual absolute difference of two angles of rotation α and β is $\min(|\alpha - \beta|, 2\pi - |\alpha - \beta|)$ (see figure 2.4). This means that in applying the distance function, one has to perform the modulo operation on its arguments in order to constrain the input angles to a half-closed interval of length 2π ($360°$), and it is also necessary to take into account the above minimum relation between the absolute difference and its explement.

2.2. The Smith–Waterman Algorithm

A well-known procedure for the local alignment of two nucleotide or amino acid sequences is the Smith–Waterman algorithm [18]. This method first assigns a similarity score to every unordered pair of elements from the alphabet (that is, the set of all nucleotides or amino acids found in the sequences). It then considers all possible alignments of the two sequences and computes their score using the weighted sum of the similarity scores of aligned nucleotide or amino acid pairs, taking insertions and deletions (commonly referred to as 'indels') into account using a certain gap penalty function. Finally, it selects the highest-scoring alignment.

The primary reason why we chose this particular algorithm is the good trade-off between its performance and accuracy. Being a dynamic programming method, Smith–Waterman runs in quadratic time (i.e., its complexity is linear in the length of both sequences), but this can be improved by heavy parallelization so that its complexity eventually becomes linear in the sum of the lengths. Even partial parallelization (which does not, strictly speaking, reduce time complexity to linear) means visible improvements in execution time. At the same time, this algorithm is proven to find the optimal alignment ('optimal' as defined by the specific scoring scheme applied). Hence, the Smith–Waterman algorithm has the potential of running significantly faster than classical, NP-hard or NP-complete structural alignment programs. Belonging to such a high complexity class, practical implementations of the aforementioned classical structural alignment methods need to avail themselves of approximations and heuristics in order to be able to finish in reasonable time. This means, however, that they might find sub-optimal alignments (e.g., by getting stuck in local extrema). In contrast, a dynamic programming approach is guaranteed to yield a globally optimal result.

It is important to note, however, that a mathematically optimal solution to a certain formulation of an alignment or comparison problem does not automatically mean that the result is biologically relevant. It is always the duty of the researcher using a computational tool to carefully evaluate the results thereof, in possession of knowledge about the complete biological context, and to verify whether or not they are correct. Naturally, this is easier if the output of the computation contains statistical indicators. These can give useful hints to the experienced scientist regarding the significance and trustworthiness of the results.

Smith–Waterman is very similar to another, earlier algorithm, the Needleman–Wunsch method [19]. However, the former is used for local alignment of strings, whereas the latter performs global alignment. For our particular use case, local alignment is preferable. It is possible to deduce from experimental data — largely found in protein structure databases such as SCOP [20] — that the secondary structure of proteins is better conserved than their 3-dimensional topology [10]. This means that we expect a local alignment to be scored more realistically, since individual features of the secondary structure, such as alpha helices and beta sheets, can be compared for common traits independently, and the local alignment algorithm will prefer to emphasize high-scoring but possibly short local similarities. In contrast, global alignment would try to force two potentially almost unrelated sequences together, collectively penalizing even strong similarities between shorter subsequences, which could result in loss of valuable data.

In our work, we modified the Smith–Waterman algorithm in several ways. First, it now operates on dihedral angles instead of the amino acid composition of proteins. Second, it applies a continuous scoring function to pairs of dihedral angles in lieu of a strict equality- or inequality-based scoring matrix. This is in accordance with our goal, since penalizing similar but unequal angles with low scores would largely counteract the attempts of the algorithm to discover otherwise hidden structural relations. Hence, a continuous function of the difference in angles seems to be a 'fairer' approach.

Just like scoring schemes based on finite matrices, our scoring function also needs at least some coarse-grained calibration. Euclidean distance might be a sensible first choice but it does not necessarily give the most realistic results. Even if the structure of the function of choice happens to correlate well with evolutionary data obtained via experiments, its parameters and constant factors still need post factum fine-tuning.

2.3. Advantages of Programmable Logic Devices

Our choice of implementation was to create specialized hardware realizing the designed algorithm. This way, no precious clock cycles are spent in superfluous operations (as would be the case with a software implementation running on generic hardware), while memory usage and energy consumption can also be optimized for the task. Furthermore, the massive parallelization required for improvements in time complexity are only practically possible using specialized hardware. The key observation about PLDs is that they allow the designer to customize the datapath in addition to the control flow, while on a fixed-architecture processor, the datapath is set in silicon.

The easiest, fastest and most available solution is to program the algorithm into an FPGA, a Field-Programmable Gate Array. This class of programmable logic devices is an industry-standard way of prototyping custom hardware, since compiling a hardware description language and programming it into an FPGA is significantly faster, cheaper, and therefore more flexible than properly fabricating an Application-Specific Integrated Circuit (ASIC) at a hardware manufacturer. The former enables quick iterations: the duration of a code-compile-test cycle is on the order of minutes or tens of minutes, while the latter usually requires months; furthermore, buying an FPGA-based development platform costs only a few hundreds or thousands of dollars (as of late 2016) and it can be reused many times, whereas the cost of ASIC production starts at tens of thousands or even millions of dollars, depending on the applied process.

2.4. High-Level Synthesis (HLS)

We have adopted the method of High-Level Synthesis as our main development paradigm. This involves implementing the desired algorithm in a high-level (usually procedural) programming language, then analyzing it and translating it to some other, lower-level form that represents hardware. In our case, we have written the 'program' in C++11, we then used the Vivado HLS toolchain to compile it to a variety of Register-Transfer Level (RTL) hardware description languages, including VHDL and Verilog.

This approach has several advantages over writing code directly at the register transfer level. First and foremost, the code is easier to write, read and understand. This makes the project more maintainable, and the essence of the algorithm becomes more apparent. In addition, there is room for low-hanging performance improvements. As the HDL representing the exact microarchitecture of the hardware is generated automatically, one can use compiler directives, flags and `#pragma`s to synthesize a wide variety of microarchitectures, and choose the one with the best performance characteristics. Automatically generating the RTL sources also has the advantage of removing certain classes of errors that could be introduced by manually changing the hardware description code (for example, when trying to find an improved microarchitecture). A related positive consequence of using a high-level language is that the code is mostly free from abstraction inversion: low-level implementation details don't influence much the main ideas in the high-level algorithm.

3. The Modified Smith–Waterman Algorithm

3.1. Synopsis of the Algorithm

As we have already mentioned, the basis of our method is the Smith–Waterman algorithm. Its basic working is as follows. Given two sequences, S_1 and S_2 of lengths L_1 and L_2, respectively, over the alphabet Σ, a matrix $M_{ij}, M \in \mathbb{R}^{(L_1+1)\times(L_2+1)}$ is filled, such that:

- $M_{i,0} = M_{0,j} = 0, \forall i \in [0, L_1], j \in [0, L_2]$
- $M_{i,j} = \max \left\{ \begin{matrix} 0 \\ M_{i-1,j-1} + s(S_{1i}, S_{2j}) \text{ (match)} \\ M_{i-1,j} + P \text{ (insertion)} \\ M_{i,j-1} + P \text{ (deletion)} \end{matrix} \right\}, \forall i \in [1, L_1], j \in [1, L_2]$

where s is the similarity score between two elements of the alphabet Σ, and P is the gap penalty (assuming a linear penalty scheme, where the gap penalty is proportional the gap length). In the above formula, "insertion" and "deletion" refer to insertions and deletions from the point of view of one of the sequences. Naturally, for the other sequence, insertions in the first one are deletions and deletions in the first one become insertions.

Trying all matches, insertions and deletions means that all possible alignments are generated, hence the global optimum (optima) is (are) always found. Setting negative scores to 0 will result in local alignments standing out.

The algorithm can optionally return the resulting alignment sequence pair itself as well. In order for this to be possible, a second, identically-sized matrix has to be filled, in which the direction of every step is recorded: rightward, downward or diagonal — depending on which neighbor of each matrix element yielded the greatest value during the computation of the max function, as described above. After the score has been computed, the element of this matrix at the index of the maximal score is used as the starting point of a backtracking procedure, whereby the path corresponding to the alignment is unrolled backwards, through the recorded 'back-pointers' to each preceding step. This backtracking requires additional memory to be allocated for the matrix that contains step directions, and it also adds complexity to the algorithm. Hence, we have avoided implementing it just yet, as we wanted to focus on simplicity and on saving silicon so that our algorithm is able to perform a quick preliminary filtering pass on large databases. For similar reasons, we decided to utilize a simpler, linear gap penalty scheme instead of a more sophisticated (and occasionally more accurate), but significantly more complex affine gap penalty that would require two additional matrices for tracking gap openings and extensions. Our solution is intended to be used as an efficient first pass performing search space reduction, in a pipeline of increasingly precise but slower and slower alignment algorithms.

In our modified variant of the algorithm, we used the following computing techniques. Real numbers representing angles are approximated using 16-bit signed integers. In order to simply and automatically take periodicity into account, we linearly projected the real interval $[-\pi, \pi)$ onto the range of integers in $[-32768, 32767]$, so that angles outside this principal interval automatically wrap around, thanks to the 2's complement representation of signed integers. The theoretically continuous scoring function is also implemented in terms of a 32-bit integer approximation. In order to save memory, not every element of the score matrix is stored, only the two sequences being aligned, and two buffers in which temporary results are kept. Finally, the computation has been partially parallelized by taking data dependencies (and the lack thereof) between matrix elements into account.

3.2. The Development Environment

For synthesizing RTL from the high-level C++ program (`align.cc`), we have used the Xilinx Vivado and Vivado HLS IDEs, and the included HLS and HDL toolchains. For writing the software driver for our alignment block and the connected AXI DMAs running on the ARM CPU, we used Xilinx's SDK. For testing in software, we compiled the HLS program and its test driver (`main.cc`) using the `Clang++` compiler and the `Make` build system, and we used Python scripts to generate pseudo-random input sets and to verify the output of our algorithm by comparing it to the result of a naive reference implementation, also written in Python. Initially, we wrote various Bash scripts for gluing these parts together and for automating batch testing of the main alignment function, `align()`. As the code evolved over time, some of these become obsolete and have subsequently been removed from the test suite. As a replacement, a verification program has been written in C that is capable of generating the necessary data sets and converting between the binary format required by the FPGA implementation and the textual format accepted by the simulation.

3.3. The Effects of HLS-related Restrictions on C++ Programs

The Physical Turing-Church Thesis states that various theoretical and physically-realized models of computation, for example Turing machines, suitable lambda calculi, combinational-sequential logic and so forth, are isomorphic and equally powerful. As a result, programs written in a Turing-complete programming language can be realized on equivalent hardware, at least theoretically. This, however, does not mean that, for example, every C++ program is directly synthesizable to hardware as-is. On our platform of choice, there are several practical implementation details resulting in restrictions that constrain the set of allowed language and library features one may use in an HLS program. This means that there will be nontrivial differences between idiomatic C++ targeted at everyday programming and C++ code that is intended for use with high-level synthesis.

For example, dynamic memory allocation is not currently possible on the FPGA (primarily due to the lack of virtual memory management) — hence most container types in the C++ Standard Library are not useful for synthesis. The main implication of this is that we have to use statically-sized arrays for storing sequence data and intermediate results, as more convenient solutions (e.g. `std::vector<T>`) would need more sophisticated memory management techniques. This in turn imposes an artificial upper bound on the size of data structures. In practice, this does not seem to be a major problem, as the distribution of protein sequence lengths is heavily biased towards shorter sequences. In fact, the vast majority of sequences is shorter than 400 amino acids. Consequently, in our implementation, we decided to make buffers of length 512, and only process sequences that fit into these. The small subset of proteins that cannot be aligned using buffers this small can be processed on, for example, a traditional desktop computer, or a modified, new version of the hardware, recompiled to use larger buffers.

Standard container types that feature dynamic memory allocation are, however, easier to debug. Memory debuggers, such as Valgrind [21], can monitor leaks and corruptions in dynamically-allocated regions more easily. Support for automatically-allocated (so-called stack) regions is usually limited or missing. Therefore, we need a way to distinguish between RTL synthesis and compilation for simulation. This is exactly the purpose of the `__SYNTHESIS__` preprocessor macro, which is defined by the Xilinx HLS compiler. We use this symbol to conditionally change the declared type of buffer variables in our code: when it is defined, buffers are assigned a synthesis-friendly primitive array type, otherwise they are declared as `std::vector` so that they can be debugged more easily.

It should be noted, however, that the use of this macro is not without risks. By its very nature, there is a possibility that the programmer makes the mistake of not writing semantically equivalent code in the two arms of the `#ifdef` directive. This can hide bugs that are only present in the synthesized piece of code but not in the one compiled for simulation. Therefore we must make sure to use this feature sparingly and to carefully examine each site of its use.

With regards to handling memory, there are even more subtle differences compared to desktop computing systems. For example, in contrast with modern, high-speed CPU-based computers, the cost and speed of accessing operative memory is comparable to the execution of primitive instructions, and thus actual computation is usually not I/O-bound. Specifically, reading and writing to distributed or block RAM on the FPGA only takes 1 or 2 clock cycles (depending on the configuration and exact usage). This means that we can — and are specifically willing to — use the relatively high bandwidth of RAM to our advantage, and invent a design that allows us to produce results as fast as the input data can physically arrive to processing elements. We still do not want to waste clock cycles waiting for memory, though, and this, together with the mode of operation of the RAM, sometimes imposes stringent requirements on the code. Specifically, the RAM blocks on the FPGA are such that each port can only be read from or written to at most once in a clock cycle. Because of this restriction, we often need to use temporary variables, lookahead buffers, shift registers and sliding windows. In a correct implementation, these constructs are materialized using registers that can be read from in parallel, by several sub-circuits simultaneously.

Another interesting side effect of such low-level memory management on the FPGA is that we know exactly what and when will be read from or written to every part of the memory. In addition, no MMU or operating system is trying to abstract away physical, direct memory addresses behind a complex virtual address handling system. As a result, out-of-bounds memory access is not fatal — addresses simply over- or underflow with respect to the address bus size. In contrast, C++ specifies that accessing memory at an invalid address (e.g. reading or writing an out-of-bounds element in an array) is undefined behavior [22], and indeed, many major compilers rely on this fact for aggressive optimization. In our code, we want to avoid writing to invalid memory locations, so checking indices and addresses is required before writing to a matrix or array element, but reading invalid data and throwing it away often simplifies the implementation quite a bit. In order to close this gap between software and hardware, the code often explicitly converts indices to unsigned if necessary, then reduces them modulo the size of the buffer they are indexing into. Buffer sizes are always chosen to be a power of two, therefore this operation has no cost at the hardware level — it is synthesized as simply ignoring the extraneous higher-order bits of the index.

As we have earlier stated, the principal purpose of designing a piece of customized hardware is to achieve better efficiency via parallelization, for example. This can be done by the programmer as well as by the HLS compiler. In the latter case, automatic parallelization consists of two, related kinds of transformation.

Using control flow and data flow analysis, the optimizer can determine which elementary operations can be performed independently, and this information is in turn used to schedule the execution of these operations in the same clock cycles, or at least to organize them in such a way that as many of the required clock cycles overlap as possible.

The synthesis tool can also pipeline longer chains of operations when explicitly asked to do so. However, this results in further requirements on language constructs usable in the code. The most conspicuous of them is the use of so-called “perfect loops.” [23] These loops need to satisfy two criteria: first, their iteration count needs to be constant, and second, in a hierarchy of nested loops, only the body of the innermost loop is allowed to contain additional nontrivial logic, besides checking and incrementing or decrementing the induction variable. This means that conventional code often needs to be rearranged, and many instances of variable initialization and computation that depend on outer loops have to be reformulated. Typically, this will result in various kinds of conditional statements inside the body of the innermost loop.

Unlike CPUs, our hardware-based solution does not realize conditional execution by jumps and by manipulating an instruction pointer. Instead, the compiler groups related conditionally-executed pieces of code into so-called blocks, and it creates an enable signal for each such block. This enable signal is then assigned the evaluated condition of the corresponding if statement, for example. This means that the size of conditionally-evaluated code groups directly affects the distribution of clock signals and the length and amount of the necessary clock wires within the circuit: the smaller the conditional blocks are, the shorter and fewer clock connections are needed, and the higher the maximal attainable operating frequency will be.

An opportunity for space optimization on the FPGA is the use of arbitrary-width buses and registers. This is supported by the synthesis tool in the form of types backed by compiler intrinsics, such as `intN` and `uintN` ($N \in [1, 4096]$), `ap_int` and `ap_fixed`. Typical C++ (and C) implementations provide only a handful of integer types, of which the width is usually 8, 16, 32 or 64 bits. This means that, for example, indexing a 512-element buffer requires the use of a 16-bit type, despite the fact that only the 9 least significant bits are actually used; this results in the 7 most significant bits being wasted. Making buffers, registers and I/O ports of the minimal width gets rid of this redundancy and it might reduce die usage. However, some of these arbitrary-precision types (notably, `intN` and `uintN`) do not compile under simulation, and even those that do would increase the complexity of the code. As such, we chose to use the standard `std::intN_t` and `std::uintN_t` ($N \in \{8, 16, 32, 64\}$) types only — the resulting hardware fit more than comfortably into the FPGA on our development board, so trying to pack everything as compactly as possible was not a real concern.

3.4. Optimization for Reconfigurable Hardware Devices

The main motivation for optimizing our code for reconfigurable hardware is the ability to parallelize it. The basic idea of parallelization is twofold. First, we observe that there are independent groups of elements in the score matrix, which is a result of its topology. Second, we argue that computing the score of a particular pair of proteins is a mathematically pure operation depending only on the two members of the pair, therefore distinct pairs can be processed in parallel. Setting up code and data so that data flow can benefit from the independence of cells is more tedious work than simply computing the alignment of several pairs at once. As a result, we first tried to satisfy the more demanding constraints of low-level, cell-wise parallelization, so that once it is properly engineered, we can more easily rely on it as a building block in the higher-level architecture thereafter.

3.4.1. Overview of the Architecture

Coarse-grained architectural design is illustrated in figure 3.1. The core of the system is the `align()` HLS function, implemented on the FPGA. This function has four inputs:

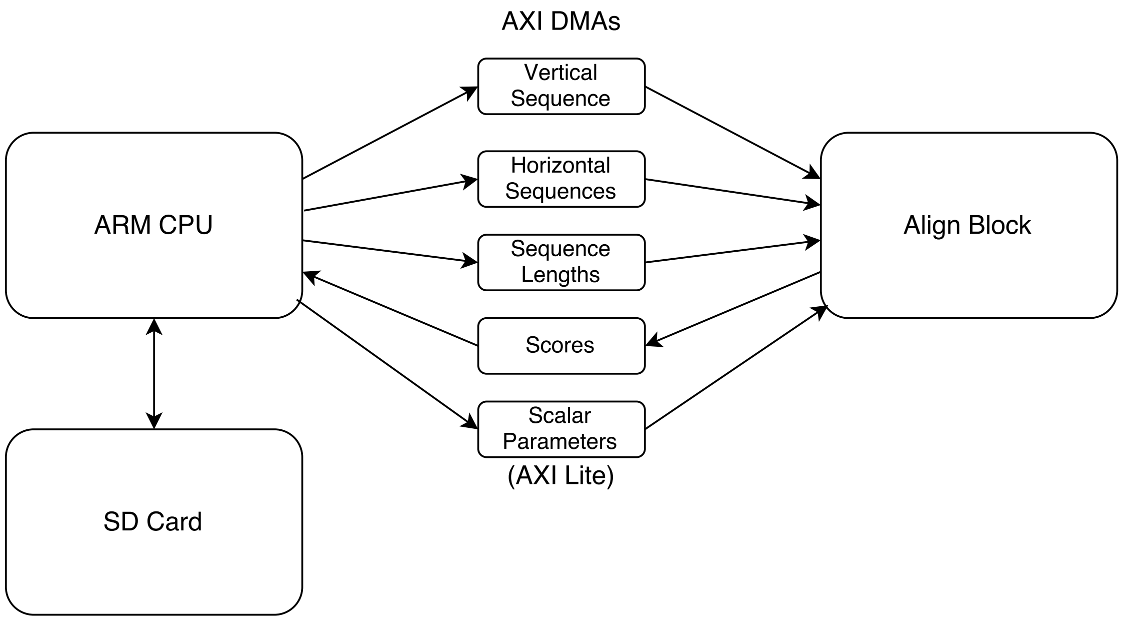

Figure 3.1. High-level architecture of the system

a static sequence (hereafter referred to as the “vertical” sequence) that is read only once and compared to every other input sequence; a list of other sequences (called the “horizontal” sequences) that the former is compared to; a stream containing the lengths of each horizontal sequence in order; and a group of scalar (i.e., not streamed) parameters. These parameters include: the length of the vertical sequence, the number of horizontal sequences, the offset parameter K of the scoring function (discussed later), and the linear gap penalty P.

The block synthesized from this function gets its inputs from the bare-metal application running on the ARM CPU; likewise, the outputs of the computation are also sent to the CPU. Communication is made possible by what is effectively memory sharing: after the bare-metal system reads the necessary data into operative memory, it initiates streaming through the three AXI DMAs located on the FPGA, between the CPU and the align block. These AXI Direct Memory Access controllers read from a specified region of RAM and transfer the bytes in chunks to the align block as it requests them. Similarly, the outputs are streamed back from the HLS block to another address in main RAM. The scalar parameters are passed to the function via the AXI Lite interface.

The input data are stored on an SD card, which is formatted as a single FAT32 partition. Although this does not result in particularly swift reads and writes, it is still adequately fast for our use case. The reasons for this are dual. Firstly, input is only read once from the card at the beginning. Secondly, calculating the scores takes remarkably longer than writing them back, so the system is not I/O bound.

It is worth mentioning why one single read suffices, though. Our initial goal was to be able to align every unique unordered pair of proteins in a database of 100,000 entries, where each sequence is at most 512 dihedral angles long. A pair of angles is described by two, 2-byte signed integers, so a sequence occupies at most $2 \cdot 2 \cdot 512$ B $= 2048$ B, therefore the size of the entire database has an upper bound of $10000 \cdot 2048$ B = 195.3 MB. An additional $100000 \cdot 2$ B = 195.3 kB of memory is required for the lengths of each sequence (also stored on 16 bits), and a maximum of $99999 \cdot 4$ B = 390.6 kB is needed by the scores of each one-to-many comparison. Hence, the overall memory usage of the data does not exceed

$$195.3\text{ MB} + \frac{195.3\text{ kB} + 390.6\text{ kB}}{1024\ {}^{\text{kB}}/_{\text{MB}}} = 195.9\text{ MB}$$

meanwhile the ZedBoard ships with 512 MB of RAM [24], so the entire database should fit into it, with quite some headroom.

Were this not the case, we would need to take advantage of different, faster ways of transferring data between the environment and the processing system. The ZedBoard features a 1000 Mbps Ethernet port [24], while other, more advanced development boards may also contain PCI Express connectivity, for increased bandwidth. The usage of these interfaces is more elaborate, though, so we have not yet considered them in our work.

3.4.2. Realization of One-to-Many Comparison in Hardware

As it should be clear from the block diagram above, the ARM processing system is in charge of initializing the configurable hardware and waiting for its answer. Each time the HLS block finishes running, it needs to be reset and restarted by the software, and this takes a non-negligible amount of time for various reasons. (I will expand on these reasons later.) Consequently, we should aim to minimize the number of such interruptions, which in turn means that the hardware shall do as much work as possible before returning control to the software.

Our approach was simple: instead of computing the alignment of just one pair of proteins, we have designed our HLS code in such a way that it compares one fixed sequence to a stream of many other, distinct sequences. The function that is responsible for this piece of functionality is `align()`; the essential parts of its body are shown in listing 3.1. (Its declaration and any Xilinx-specific compiler directives are omitted for brevity.)

Listing 3.1. One-to-many alignment

```
typedef std::int16_t index_type;
typedef std::uint16_t size_type;
// ...
bool should_read_ver_stream = true;

for (seq_count_type i = 0; i < num_streams_hor; i++) {
  index_type stream_size_hor = stream_sizes_hor.read();

  // Compute score
  score_type score = align_one(
    stream_ver,
    stream_size_ver,
    streams_hor,
    stream_size_hor,
    scoring_offset,
    gap_penalty,
    should_read_ver_stream
  );

  // Add AXI side-band signals to raw numeric value
  // Set TLAST so that partial bursts are flushed
  // Set keep and strobe signals to all 1's
  auto axi_score = axi_out_score_type{};

  axi_score.data = score;
  axi_score.keep = -1;
  axi_score.strb = -1;
  axi_score.last = i == num_streams_hor - 1;

  out_scores.write(axi_score);

  // only read vertical stream once
  should_read_ver_stream = false;
}
```

The `align_one()` function (listing 3.2) is the subroutine that actually performs the alignment algorithm on two sequences. Its last Boolean parameter, `should_read_ver_stream` determines whether it reads from the vertical stream. This parameter is set to `true` during the first iteration, and to `false` otherwise. When the function does read from the stream, it also also stores the acquired data in a local, `static` buffer. It then reuses this buffer during subsequent iterations.

Listing 3.2. Declaration of `align_one` and related types

```cpp
typedef std::int32_t score_type;
typedef std::int16_t angle_type;

struct Dihedral {
  angle_type phi;
  angle_type psi;
};

score_type align_one(
  hls::stream<Dihedral> &stream_ver,
  index_type            stream_size_ver,
  hls::stream<Dihedral> &stream_hor,
  index_type            stream_size_hor,
  score_type            scoring_offset,
  score_type            gap_penalty,
  bool                  should_read_ver_stream
);
```

It is also worth noting the element type of the output stream. Unlike the input parameters, which contain primitive types (integers) directly, the `out_scores` stream manages objects of type `axi_out_score_type`, which is a `typedef` for `ap_axis<sizeof(score_type) * CHAR_BIT, 1, 1, 1>`.

This is necessary because the side-band signals, `keep`, `strb` (strobe) and `last` are needed for correct operation of the AXI DMA. All bits of `keep` and `strb` need to be set to high in order for the DMA to send every byte as per the request of the software controller. The `last` bit should be asserted upon the last iteration, since the DMA might only flush complete bursts, but the last burst is partial when the size of the results is not a multiple of the burst size, so it might be lost without the appropriate setting of this control bit.

3.4.3. Low-level Implementation of One-to-One Comparisons

Parallelizing the computation of elements in a score matrix is made possible by the observation that data dependencies between such matrix cells are monotonic, as indicated in figure 3.2. The value of a cell only depends on the value of its top, left and top-left diagonal neighbors. Consequently, cells on the antidiagonals are independent of one another, and their value can be obtained in parallel.

The de facto standard for parallelization of sequence alignment on configurable hardware is the linear systolic array or LSA [25]. The essence of this architecture is a homogeneous set of closely-connected, uniform nodes forming a linear directed graph, where every node receives input from its predecessors, transforms it, and sends it on to its successors. The modus operandi of the LSA as described in [26] is the following.

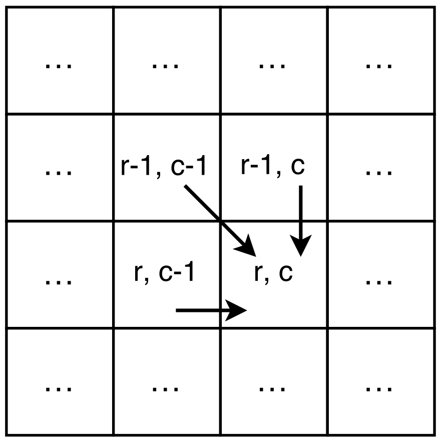

Figure 3.2. Data dependencies within the score matrix

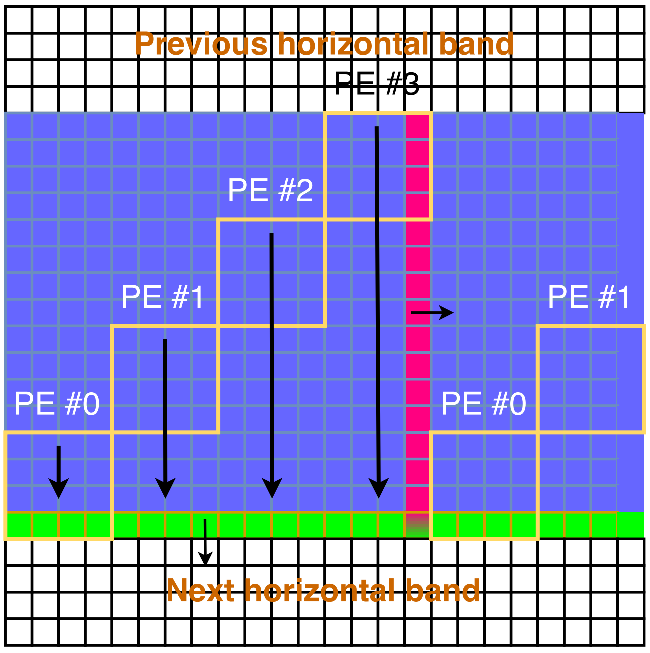

Figure 3.3. Operation of the Linear Systolic Array

The score matrix is divided into several horizontal bands. Each horizontal band is then subdivided into vertical columns, and every such column is assigned a processing element (PE) of the LSA. Each PE works with only one rectangular window of its column at a time. The first PE begins traversing the leftmost column of the horizontal submatrix, starting in its top left corner. When it finishes computing all partial results in the rectangular window, it moves down by one window. It also passes the partial results of the bottom row of the window downward so that its future self can use them as input one step later, in the next window. It propagates the results in the rightmost column of its window to the right as well, where the second PE picks them up and starts processing the first rectangular window of the second column, with a one-step delay. There are additional streams and FIFOs that transport temporary results between bigger parts of the matrix, e.g. between two horizontal bands. For better understanding, the working of this system is illustrated in figure 3.3.

It should be obvious at this point that this solution is quite complicated. As a result of hierarchically subdividing the matrix on many levels, several different boundary conditions need to be handled. This means that changing or extending the architecture may take lots of effort, and it also has the potential of introducing subtle bugs into the logic.

A simpler design is outlined in [27]. This is based on the fact that the overwhelming majority of proteins is at most a few hundred amino acids long [1] [2]. Hence, it is not necessary for us to come up with a completely general solution that is capable of handling sequences of unbounded length. This can spare us some levels of matrix subdivision, simplifying the code substantially.

The cornerstone of this architecture is an array of smaller, subordinate processing units inside the top-level PE, each of which computes the value of only one cell in an antidiagonal. Again, the computation starts at the top left corner of the score matrix and descends by one antidiagonal in every iteration.

Even this approach is not without weaknesses, though. Processing every element of an antidiagonal in parallel means that every one of their intermediate results needs to be stored simultaneously, so the memory requirement of the system is high. Specifically, it is proportional to the length of the longest antidiagonal, which is equal to the length of the shorter of the two sequences being aligned. Moreover, a major part of the allocated memory is wasted because it is only when the longest antidiagonals are being computed that it is completely occupied. In practical implementations, the maximal number of rows and columns is equal, so the biggest allowed score matrix is rectangular. This means that it only has one longest antidiagonal, so it is possible that the memory storing partial results is only ever fully occupied in one iteration. Equally importantly, most of the RAM also remains unused when aligning two short sequences.

We have ameliorated this architecture by partitioning the matrix into vertical strides of 16 columns each. The cells of each such stride are traversed and computed just like previously, and the partial results generated in the last column thereof are propagated horizontally, to the right, with the aid of a dedicated buffer that we call the horizontal propagation buffer. Figure 3.4 depicts this improved topology.

Filling the matrix in multiple steps demands changes in other aspects of the design, too. For instance, it is no longer possible to read the horizontal sequence all at once, as it would be the case, had we not partitioned the matrix. During the iteration over each stride, a new 16-element fragment of this sequence should be read, with the exception that the last fragment may be shorter if the length of the sequence is not divisible by the stride width. The code that handles partitioning is displayed in listing 3.3.

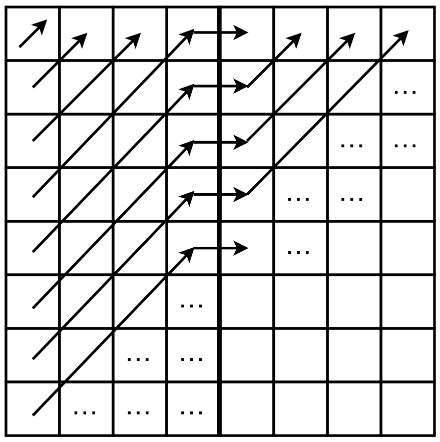
Figure 3.4. Vertical subdivision of the score matrix into strides of width 4. The horizontal propagation buffer is represented by the thick line in the middle.

Listing 3.3. Vertical partitioning scheme

```
static Dihedral seq_ver[WIN_ROWS];
static Dihedral seq_hor[WIN_COLS];
score_type hor_prop_buf[WIN_ROWS];

index_type h;

#define MAX_SEQ_SIZE      512
#define WIN_COLS          16
#define WIN_COUNT_HOR     (MAX_SEQ_SIZE / WIN_COLS)
// ...

hor_window_loop:
  for (h = 0; h < WIN_COUNT_HOR; h++) {
    // ...
    // (computation of each antidiagonal happens within this loop)

    // (after the score of the last column, ‘cur_score’ is computed)
    // propagate cells in the last column of each row rightwards
    if (c == WIN_COLS - 1) {
      if (in_bounds) {
        hor_prop_buf[r] = cur_score;
      }
    }
  }
```

The actual traversal of a vertical streak is based on a simple linear transformation. Let N and K be the number of rows and columns of the streak, respectively, and let (r, c) be the 0-based row and column indices of the same element. We then define the transformed

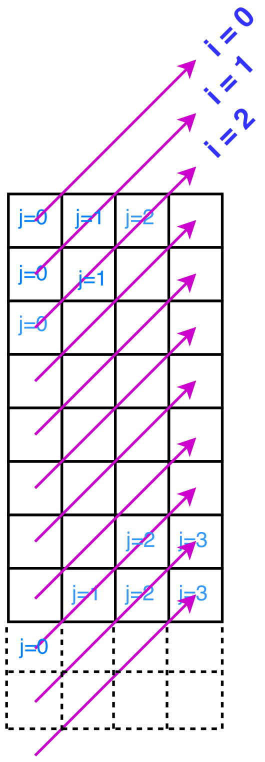

Figure 3.5. Transformed coordinates in an 8×4 vertical streak. Some elements are out of bounds, because they are either above the first row or below the last one.

coordinates (i, j) in a way that they satisfy the following system of equations:

$$\begin{cases} r = i - j \\ c = j \end{cases} \tag{3.1}$$

From this, we can see that i is the (0-based) index of the antidiagonal within the streak (the top left antidiagonal has index $i = 0$), and j is the index of the element within the antidiagonal itself, with a twist. When $i < N$, the bottom left element of each antidiagonal has index $j = 0$. When $i \geq N$, the first $N - i$ elements of subsequent antidiagonals extend beyond the streak, “sticking out” downward, so they do not designate valid matrix elements. This is illustrated in figure 3.5.

With these definitions in mind, we can now describe the relationship between elements. Namely, the left neighbor of the element at coordinates (i, j) is located at $(i - 1, j - 1)$, its top neighbor is at $(i - 1, j)$, and its top left neighbor has coordinates $(i - 2, j - 1)$. This change of basis is required because parallelization forces us to enumerate the matrix elements by iterating over antidiagonals. By introducing this trick into the indexing scheme, it is not necessary to write separate indexing code for the two cases, and the coordinates of the neighbors are constant (relative to a given cell). We can then simply drag a sliding window (two-dimensional shift register) over a group of three neighboring antidiagonals without having to worry about the separation of cases.

Of course, it is necessary to take care of underflowing elements, but that is as easy as checking if $0 \leq r < N$. This condition is captured in the variable `in_bounds`. Furthermore, bounds checking would have had to be done anyway, since some of the first antidiagonals also extend beyond valid coordinates, but in the opposite direction. This can be easily shown by considering that the number of antidiagonals in the streak is $N + K - 1$, so $i \in [0, N + K - 1)$ and $j \in [0, K)$. By substituting appropriate values of i and j into equation 3.1, it is possible that either $r < 0$ or $r \geq N$, and that is exactly how out-of-bounds indexing occurs.

Another kind of boundary condition concerns cells in the first row or first column of each stride. Cells in the leftmost column ($c = 0$) have no left and top-left neighbors; the values of these virtual neighbors must be read from the corresponding elements of the horizontal propagation buffer. Additionally, in the very first vertical stride, when this buffer is not yet initialized, the cells are defined to have all zero neighbors. Cells on the top of the stride ($r = 0$) lack a top and top bottom neighbor; the algorithm defines that the values of these missing neighbors should also be 0. Code handling coordinate transformations and boundary conditions is listed in 3.4.

Listing 3.4. Change of basis and boundary conditions

```
index_type i, j, r, c;
index_type max_valid_row, max_valid_col;
diag_loop:
for (i = 0; i < WIN_DIAGS; i++) {
  col_loop:
  for (j = 0; j < WIN_COLS; j++) {
    r = i - j;
    c = j;
    // ...
    in_bounds = 0 <= r && r < WIN_ROWS;
    // Read in horizontal sequence buffer at the beginning of each stride
    if (r == 0) {
      if (stream_size_hor == 0) {
        if (c < max_valid_col) {
          max_valid_col = c;
        }
      } else {
        seq_hor_read_reg = stream_hor.read();
        seq_hor[c] = seq_hor_read_reg;
        stream_size_hor--;
      }
    }
  }
}
```

As described above, the dependencies of each matrix element are located in two different antidiagonals. Considering an element at index (i, j), two of its neighbors are located in the antidiagonal at index $i-1$, and the third one is found in the one at index $i-2$. From this, one might deduce that three buffers are necessary: two that contain the neighbors, and one in which the results are written. However, with some lookahead, it is possible to work with two buffers only. The lookahead is performed by the aforementioned 2×2 sliding window, `lah_buf` (figure 3.6). At the beginning of each iteration over an antidiagonal, this window is filled with 4 values so that it contains all three neighbors of the first cell, while the two buffers, `diag_buf_old` and `diag_buf_new`, contain the elements of antidiagonals at indices $i-1$ and $i-2$, respectively. For computing the score of a cell, values are only ever taken from the sliding window. This means that the appropriate element of `diag_buf_old` can be overwritten by the corresponding element of `diag_buf_new`, then the newly-computed score can be written back to `diag_buf_new` at the same index. In this manner, the contents of the two antidiagonal buffers are replaced gradually, as if in a circular buffer, and no third temporary array is needed. The code responsible for this piece of functionality is presented in listing 3.5.

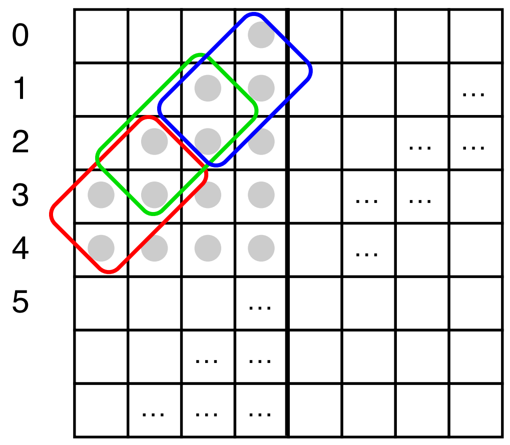

Figure 3.6. The sliding window, running over a 4-column stride. The red, green and blue windows contain the neighbors of the 1st, 2nd and 3rd element of the 5th antidiagonal, respectively. The 0th element has some out-of-bounds neighbors, so it is omitted here for the sake of simplicity.

Listing 3.5. Code for the sliding window

```
// +------+------+
// |(0, 0)|(0, 1)|
// +------+------+
// |(1, 0)|(1, 1)|
// +------+------+
//
// the ARRAY_PARTITION directive ensures that lah_buf is
// implemented with four registers, not as block RAM
score_type lah_buf[2][2];
#pragma HLS ARRAY_PARTITION variable=lah_buf complete dim=0

// ...

// inside the innermost loop, 'col_loop':
if (j == 0) {
  lah_buf[0][0] = i < 1 ? 0 : hor_prop_buf_next_cells[0];
  lah_buf[1][0] = i < 0 ? 0 : hor_prop_buf_next_cells[1];
} else {
  lah_buf[0][0] = lah_buf[0][1];
  lah_buf[1][0] = lah_buf[1][1];
}

// Read ahead, respecting boundary conditions.
// Out-of-bounds values should be 0, so that 'partial'
// strides (occurring at the end of sequences  when the length
// is not an integer multiple of 16) will have correct values on
// the boundaries (since not all invalid elements of a partial
// stride are out of bounds!)
diag_buf_old_next_cell = j <= i - 2 ? diag_buf_old[j] : 0;
diag_buf_new_next_cell = j <= i - 1 ? diag_buf_new[j] : 0;
lah_buf[0][1] = i < 2 ? 0 : diag_buf_old_next_cell;
lah_buf[1][1] = i < 1 ? 0 : diag_buf_new_next_cell;

// ...
// The computation of the score of the current cell uses lah_buf:
cur_score = array_max<score_type, 4>({
  lah_buf[0][0] /* diag neighbor */ + dihedral_score(
    seq_ver_comp_reg,
    seq_hor_comp_reg,
    scoring_offset
  ),
  lah_buf[1][0] /* left neighbor */ + gap_penalty,
  lah_buf[1][1] /* top  neighbor */ + gap_penalty,
  score_type(0) /* align locally */
});
```

This is not the only occasion when a temporary register must be used, though. The dual-port RAM inside the FPGA can perform only one read and one write operation in the same clock cycle. Therefore, when the code needs to reuse some data that is located in RAM, it must first read it into a temporary register for efficiency. Registers can be read from by many components of the circuit at the same time, and they also retain their value, so accessing them multiple times either in parallel or sequentially is possible without a performance penalty. In the (special) case where partially overlapping ranges are read from an array in two consecutive iterations, this usually manifests in a need for a shift register: the new non-overlapping element(s) is (are) pushed onto the shift register, while the common items remain in it, shifted by the suitable number of places (usually one).

In short, the modifications presented hereby reduce the amount of operative memory as follows. Given a maximal sequence length of 512, the "naïve", non-partitioning solution would need:

- two buffers for storing the entirety of both sequences, and
- two other buffers capable of holding the intermediate results on the longest antidiagonal.

As both scores and pairs of dihedral angles are stored on 4 bytes, this adds up to a total of $2 \cdot 512 \cdot 4\ \text{B} + 2 \cdot 512 \cdot 4\ \text{B} = 8192\ \text{B}$ per top-level processing element. In contrast, the improved, partitioning method requires:

- one buffer of size 512 for storing the vertical sequence;

- another buffer of size 16 for the fragment of the horizontal sequence being processed;
- a horizontal propagation buffer of 512 elements to pass on partial results to the next stride; and
- the two antidiagonal buffers of size 16

for a total of $512 \cdot 4\text{ B} + 16 \cdot 4\text{ B} + 512 \cdot 4\text{ B} + 2 \cdot 16 \cdot 4\text{ B} = 4288\text{ B}$. Altogether, the memory saving introduced by our transformation is

$$\frac{8192\text{ B} - 4288\text{ B}}{8192\text{ B}} = 0.4766$$

or approximately 48%.

As a quantitative measure of similarity of dihedral angles, we use the following scoring function:

$$s(\Phi_1, \Psi_1, \Phi_2, \Psi_2) := K - [d^2(\Phi_1, \Phi_2) + d^2(\Psi_1, \Psi_2)]$$

where $d(\alpha_1, \alpha_2) := \min(|\alpha_1 - \alpha_2|, 2\pi - |\alpha_1 - \alpha_2|)$ and $K \in \mathbb{R}^+$. The constant K is called the "offset". Along with the linear gap penalty, P ($P \in \mathbb{R}^-$), they form the parameter space (K, P) of the alignment algorithm.

3.5. Challenges of Debugging and Testing

While implementing and testing the C++ code, we have run into many design difficulties and bugs. Sometimes, it took several hours of hard work to discover, debug and fix them. The most frequent and typical mistakes can be classified into four categories.

The first category consists of "classic" low-level programming mistakes, such as uninitialized variables and off-by-one errors. Most of these were caught by the compiler even before running the simulation: `clang++` performs extensive control flow and data flow analysis, so it is able to warn about a wide variety of locally-detectible data flow violations. The notable exception was a set of off-by-one errors that were mainly caused by improper initialization of shift registers. For example, comparison of the debug output of an early version of the `align()` function to that of the reference implementation showed a discrepancy in the partial results obtained from the second and subsequent vertical streaks. Interestingly, partial results of the first one were always correct. It turned out that an off-by-one error in the initialization of the horizontal propagation buffer accounted for this problem, that did not affect the matrix elements in the first streak, since in this area, left neighbors of the first matrix column are not read from the horizontal propagation buffer.

The second group of errors arose from complications introduced by the constraints necessary for achieving perfect loops. As it has already been noted, these resulted in many interacting conditionals that simultaneously affect state as well, severely increasing the cyclomatic complexity of the innermost loop. The prime example of this is the code that decides whether to read from the input streams and checks if data read from the internal buffers is valid. In particular, due to an incorrect setting of the `max_valid_row` and `max_valid_col` variables, the invalid partial results from the first completely unused vertical stride were not ignored. This issue was remedied by specifying that these two variables are only ever allowed to decrease, not increase. This is correct, because their value is initially the maximal width and height of a vertical stride (16 and 512, respectively), then they drop to one past the row and column indices of the last valid element within the stride, then `max_valid_col` is set to 0 in the first entirely invalid vertical streak, if any.

Misunderstandings of APIs provided by the HLS toolchain and the Board Support Package for the ARM system accounted for bugs in the third category. For instance, an early revision of the code used the `hls_stream::empty()` method for checking whether there were remaining dihedral angle pairs in the input streams. This appeared to work correctly in the simulation, but it failed when programmed into the FPGA. The reason for this is that the `empty()` method returns `true` when the DMA is not able to deliver input data at the rate requested by the processing element, even if there are items left in the stream. This bug was corrected by explicitly specifying the length of each stream as additional arguments to the `align()` function.

Another mistake related to API usage was discovered while interfacing with the XilFFS library, provided as part of the Board Support Package for the ARM bare metal system. It turns out that some of the functions in this library are designed very unconventionally and they are implemented in a way that encourages bad coding style. Specifically, reading from the SD card requires mounting it first, and the output of the `f_mount()` function, as an out parameter, is a pointer to the filesystem descriptor object corresponding to the mounted volume. This object appears not to be used anywhere else; in particular, the `f_open()` function does not refer to it in its parameter list. This made us assume — given that the documentation failed to state otherwise — that this out parameter is safe to throw away.

Examining the libxilffs code reveals that the intention of the library authors was to allow DOS- and Windows-style drive specifications in file paths, such as `0:/INPUT.BIN`. This minor convenience feature made the authors design `f_open()` (and other functions) in such a way that instead of referring to the filesystem object directly, the function extracts the volume number from the path (or uses a default of 0), then uses it to index into a global array of pointers. This global array is filled by `f_mount()` with the pointer to the filesystem object passed in as its out parameter. Therefore, there is implicit global state in many file manipulation functions, as they all depend on the filesystem descriptor being kept around by the user as long as he or she is using the FFS library.

This misleading behavior was not immediately apparent in the official example code, as its authors declare numerous important variables as `static` globals [28]. We consider this programming style controversial and questionable at best, therefore we opted for minimizing the lifespan of variables (including the filesystem descriptor object) instead. This results in a seemingly unused local variable for the filesystem object (which is only written to but never read from) in the code, but we feel that this approach of dealing with a misleading interface is still the lesser of two evils.

The fourth group of bugs was related to the general tendency of hardware to manipulate data in bigger chunks. For example, handling the FAT filesystem revealed yet another peculiarity. Files of a size not evenly divisible by the maximal sector size were truncated because the last partial sector was not written. For files smaller than 512 bytes, this meant that no data was written at all, the output file was empty, despite `f_write()`, `f_sync()`, `f_close()` and `f_mount()` (used for unmounting) having returned `FR_OK`. Therefore, explicitly synchronizing and closing the file then unmounting the filesystem did not help. We solved this issue by padding all data to be written to a file with trailing zeros, so that it had a length that is a multiple of the maximal sector size. The format in which the resulting scores are stored contains the number of sequences anyway, which makes it easy to compute the number of scores. (If the number of sequences is N, then the number of scores is $\frac{N \cdot (N-1)}{2}$, as every sequence is compared with every other one exactly once, and the comparison function is symmetric.) Therefore, the need to append padding to the end of files did not affect semantics and correctness.

Another similar error occurred in connection with the AXI DMA responsible for transmitting the sizes of horizontal sequences to the alignment block. The type used to represent the length of each sequence is `index_type` that is currently defined as a signed 16-bit integer. The minimal bus width of AXI DMAs on this platform is 32 bits, consequently, they require all data to be aligned to 4-byte boundaries by default. In the code running on the ARM CPU, all sequence sizes are allocated in one contiguous array. Each time a new alignment is initiated, the next sequence in order will become the "vertical" one, while subsequent ones will serve the role of the "horizontal" sequences, so the pointer to the array in which sequence lengths are stored must be incremented by one. Because `sizeof(index_type) == 2`, this means that the pointer will not always be 4-byte aligned, which initially resulted in the DMA aborting with an error. It would have been possible to always allocate a new buffer dynamically (`malloc()` is always guaranteed to return a pointer that is sufficiently aligned for any memory operation), but that would have resulted in superfluous copying and an "accidentally quadratic" running time. Instead, the problem was solved by configuring this particular DMA to allow unaligned reads. This makes the device somewhat more complex, so it requires more space on the FPGA, but in our opinion, the speed gain is worth the increased hardware usage in this case.

4. Results and Evaluation

4.1. Correctness of Simulation and Hardware

To the best of our knowledge, all bugs found by testing or by inspection of the code have been fixed in both the simulation and the synthesized logic circuit. We have verified this by generating random sequences using the `multi_gen_random_seqs.c` tool. One such collection of example inputs can be found in the attached `INPUT.BIN` file. We have then compared the output of the simulation (i.e. the C++ code in `align.cc` running on a computer, wrapped in a thin driver, `main.cc`), that of the synthesized hardware (the same C++ code running inside the FPGA), and that of a naïve reference implementation written in Python (`smith_waterman.py`). All three systems have been initialized with the same parameters, namely $K = 65536$ and $P = -4000$. For all provided inputs, they have generated consistent output, so we believe that they are working correctly.

4.2. Performance of the Synthesized Logic

We have already contrasted the relatively low — quadratic — asymptotic time complexity of the Smith–Waterman algorithm with the high — generally combinatorial or exponential — complexity of NP-hard structural alignment and comparison methods. In this section, we assess its performance with special regards to time efficiency, effective hardware use, dissipated power and die usage.

4.2.1. Physical Execution Time

Initially, a target clock frequency of 100 MHz has been set in the HLS project. At first glance, this did not seem to have resulted in sufficiently conservative pipelining: the pipeline created by the synthesis tool had a depth of 8, and the tool's estimation for the minimal clock interval was 10.27 ns, which implies a peak clock frequency of $\frac{1}{10.27 \text{ ns}} = 97.37$ MHz. Even so, when we imported the generated HDL into Vivado, synthesis, implementation and bitstream generation succeeded, meaning that the toolchain analyzed the performance of the code and concluded that it is able to reliably sustain a clock rate of 100 MHz. This ostensible contradiction can be explained by the fact that high-level synthesis gives only a rough estimation of the timing parameters of the circuit, and it assumes worst-case environmental parameters, such as high die temperature and low power supply voltage.

Nevertheless, we tried to force the synthesis tool to apply more conservative engineering margins, and we did so by specifying a target clock period of 5 ns. This seems to have

Computer	32.3 s
FPGA	44.7 s

Table 4.1. Absolute net execution time on a computer and on the FPGA

helped, and the pipeline of the new microarchitecture became deeper — it contained 13 elements, so the compiler was able to expand the computation, and less work was expected to be done in a single clock cycle. The end result was a minimal clock period of 8.46 ns, corresponding to a nominal maximal clock frequency of 118.2 MHz. This variation can be rightfully expected to work reliably with the current 100 MHz setting, and later it will even be possible to use the PLL and the frequency divisors on the development board to raise this frequency to no less than 118.18 MHz. This is the highest value not exceeding 118.2 MHz that can be derived from the built-in 100 MHz clock using the PLL and the two 5-bit frequency divisors. The latter two would need to be set up so that they provide an upscaling ratio of $\frac{13}{11} = 1.\overline{18}$.

The target initiation interval (II) of 1 has been achieved at first attempt, so the processing element is able to produce one partial result (matrix element) in every clock cycle. Naturally, bubbles formed in the pipeline during the transition between two vertical bands of the matrix decrease the overall efficiency, but the approximation based on the clock frequency and the II, $\frac{100\text{ MHz}}{1} = 10^8\ {}^1\!/\!_\text{s} = 10^8$ CUPS, is a reasonable upper bound on the performance of the device.

For comparing the actual average performance of the hardware to that of the simulation, the latter has been compiled with debugging and logging turned off, at a high optimization level (`-O3 -flto`). Both drivers — the one in the simulation and the one running on the ARM-based processing system — contain code that is carefully crafted so that it only measures and accumulates time actually spent processing, so it excludes reading from file or from standard input, writing scores to a file or to the serial port, setting up the alignment system between restarts, and basically everything identified as noise.

The simulation has been run on a mid-2014 MacBook Pro featuring 16 GB of RAM and a 2.2 GHz Intel Core i7 CPU with 4 cores. Both implementations have been fed the same set of inputs (specifically, those contained in the aforementioned `INPUT.BIN` file). Using a modified version of the code of the simulation, we have calculated that the total number of cells computed during the comparison of each pair of sequences is 2 257 196 091. This does not include throw-away cells, i.e. those that are out of bounds and contain invalid data but that the code still computes for reasons of convenience and performance, as already explained in the low-level architectural description.

Based on many measurements, the time spent computing the results was very stable, it had little variance. In fact, when rounded to 3 significant figures, it was always the same on a specific device, as presented in table 4.1.

This shows that a single processing element could not quite beat the more powerful CPU on its own.

Device	Computer	FPGA
Total cells computed	2 257 196 091	2 257 196 091
Effective computation time	32.3 s	44.7 s
Clock frequency	2.2 GHz	100 MHz
Cell updates per second (CUPS)	$6.988 \cdot 10^7$ 1/s	$5.050 \cdot 10^7$ 1/s
Clock cycles per cell	31.48	1.985
Relative performance (lowest = 1)	1.000	15.86

Table 4.2. Average number of clock cycles spent on one cell update

4.2.2. Effective Hardware Use

It is not only absolute quantities that matter, though. It is interesting to calculate certain relative measures too. For example, a good indicator of effective hardware use is the mean number of clock cycles necessary for computing a cell. This value is shown in table 4.2. We can infer that the general-purpose CPU uses more than 15 times as many clock cycles as the FPGA does for accomplishing the same task. This means that the specialized hardware is better able to utilize the available resources, which is in accordance with our predictions.

It is also noteworthy that in spite of an initiation interval of 1, the hardware needs nearly two clock cycles on average for one cell update. This inefficiency can be attributed to the need for emptying and refilling the pipelines between vertical strides of the score matrix. The reason they must be fully emptied after each streak is that the synthesis tool was not able to "rewind" the pipelines because the two outer loops of `align_one()`, namely, `hor_window_loop` and `diag_loop`, are not so-called perfect loops. They contain conditional `break` statements that interrupt them after reaching the end of the horizontal and vertical stream, respectively. These are necessary because it would be prohibitively inefficient to continue spinning until each and every element of the maximally-sized, 512×512 matrix is computed — most of them invalid — even when the sequences are short. By and large, we are trading off a big slowdown for a smaller one.

4.2.3. Dissipated Power and Energy

We have measured the power usage of both the CPU and the ZedBoard. On the computer, we have used Battery Health [29], a software for monitoring various aspects of power and energy usage. The measurement was conducted by taking a note of the estimated power input displayed by Battery Health before running the software simulation, then subtracting it from the value displayed while the simulation was performing alignments. This experiment has been repeated 9 times. The results of the measurements are shown in table 4.3.

The two lowest and the two highest values are so disproportionately small (3.9 W, 5.6 W) and large (12.3 W, 13.4 W), respectively, that we concluded there was a mistake in the measurements that yielded these four values. Hence, we disregarded them when calculating the mean and the standard deviation of the sample.

The result is that the average power consumption of the simulation running on the

Power while running (W)	Power while idle (W)	Power used for alignment (W)
25.7	18.9	6.8
30.3	18.0	12.3
27.7	18.7	9.0
24.8	17.2	7.6
25.9	20.3	5.6
28.8	20.4	8.4
30.2	16.8	13.4
23.4	19.5	3.9
29.2	20.5	8.7

Table 4.3. Power measurements on the CPU

Current, active (A)	Current, idle (A)	Difference (A)	Net Power (W)
0.28	0.20	0.08	0.9792
0.28	0.19	0.09	1.1016
0.29	0.20	0.09	1.1016
0.28	0.20	0.08	0.9792
0.28	0.19	0.09	1.1016

Table 4.4. Power measurements on the FPGA

CPU is $P_{CPU} = 8.1\text{ W} \pm 0.894\text{ W}$.

On the ZedBoard, the amount of dissipated power can be computed from the voltage of the power supply and the current the device draws from it. The voltage was measured directly using a digital multimeter, and we found that it was $V_{DD} = 12.24\text{ V}$. In contrast, the current can only be measured indirectly, with the help of two dedicated current sense pins, designated by the identifier `J21` on the PCB. According to the ZedBoard Hardware User's Guide, there is a 10 mΩ resistor connected to V_{DD} in series with the rest of the circuit, and the current sense jumpers are wired to its pins. By measuring the voltage drop across this resistor, we can calculate the current and the power consumption of the circuit. As with the measurements on the computer, we have probed the current sense pins twice: once before launching the computation, and once while it was in progress. No values were excessively small or large, so we have only performed 5 measurements and used all of them for computing the mean and the standard deviation. The data is presented in table 4.4. The estimated average power consumption of the hardware is thus $P_{FPGA} = 1.053\text{ W} \pm 0.067\text{ W}$.

The Vivado system also calculates an approximation of what it calls "static" and "dynamic" power. The former is the power drawn by the SoC when it is in standby, the latter is the difference between the active power and static power, so the total power is the sum of these two quantities. The values, displayed in the Project Summary by Vivado, are repeated in table 4.5 for comparison with the measured power consumption.

Thus, the ratio of the average power consumption of the CPU and that of the ZedBoard is $\frac{P_{CPU}}{P_{FPGA}} = \frac{8.1\text{ W}}{1.053\text{ W}} = 7.69$.

We can also compute the energy used by the two systems. For the CPU, this is $E_{CPU} = 32.3\text{ s}\cdot 8.1\text{ W} = 261.6\text{ J}$; for the FPGA, it is $E_{FPGA} = 44.7\text{ s}\cdot 1.053\text{ W} = 47.07\text{ J}$.

Static Power	0.160 W
Dynamic Power	1.602 W
Total On-Chip Power	1.762 W

Table 4.5. Power estimations from Vivado

Component	Used	Available	Utilization (%)
Slice LUTs	7322	53200	13.76%
Slice Registers	8998	106400	8.46%
F7 Muxes	1	26600	<0.01%
Slice	2873	13300	21.60%
LUT as Logic	6856	53200	12.89%
LUT as Memory	466	17400	2.68%
LUT Flip Flop Pairs	9177	53200	17.25%
Block RAM Tile	11.5	140	8.21%
DSPs	2	220	0.91%

Table 4.6. Hardware utilization on the FPGA

The ratio of the energy consumptions is thus $\frac{E_{CPU}}{E_{FPGA}} = \frac{261.6\text{ J}}{47.07\text{ J}} = 5.558$. This means that for the given task, the FPGA solved the problem more than 5 times more energy-efficiently.

4.2.4. Die Usage

Assessing what fraction of the FPGA fabric is occupied by the synthesized circuit is essential for predicting the scalability of the system, as it limits the number of parallel processing elements one can create on the chip. Fortunately, the Vivado IDE comes to our aid by providing useful reports. When routing — the last step of implementation — is complete, we can open the Implemented Design item, then click on Report Utilization. The generated document (partially replicated in table 4.6) contains a detailed list of the utilization of each kind of logic element.

It is quite apparent that the bottleneck is in the number of slices. Almost $\frac{1}{4}$ of these elements is used, so if we duplicated the synthesized code as-is, it would fit in the chip 4 times. Of course, when we create an actual parallelized variant out of the hardware, only the processing elements performing the alignment need to have multiple copies, whereas other IP cores, such as the Zynq 7000 Processing System and the AXI DMAs, will have only as many instances as there currently are. Therefore, we expect that more than 4 independent PEs can be created, but for the sake of conservative estimation, we will keep this lower bound in mind.

However, it is also worth observing the hardware utilization of each individual block on its own. Using these numbers, we can tell more accurately how many of them could be placed on this particular FPGA. For example, the `align()` block needs only slightly over 6% of available slices, while the second largest block, the AXI DMA interconnect, takes up approximately 5% of the surface, as presented in tables 4.7 and 4.8.

Based on these numbers and the fact that slices always have the highest usage, we can carry out the following back-of-the-envelope calculation. A total of 2873 slices are

Component	Used	Available	Utilization (%)
Slice LUTs	2375	53200	4.46%
Slice Registers	2531	106400	2.38%
F7 Muxes	0	26600	0.00%
Slice	837	13300	6.29%
LUT as Logic	2296	53200	4.32%
LUT as Memory	79	17400	0.45%
LUT Flip Flop Pairs	2691	53200	5.06%
Block RAM Tile	1.5	140	1.07%
DSPs	2	220	0.91%

Table 4.7. Logic elements used by the `align()` block

Component	Used	Available	Utilization (%)
Slice LUTs	1680	53200	3.16%
Slice Registers	2042	106400	1.92%
F7 Muxes	0	26600	0.00%
Slice	698	13300	5.25%
LUT as Logic	1553	53200	2.92%
LUT as Memory	127	17400	0.73%
LUT Flip Flop Pairs	2165	53200	4.07%
Block RAM Tile	0	140	0.00%
DSPs	0	220	0.00%

Table 4.8. Logic elements used by the AXI memory interconnect

occupied, 837 of which make up the align IP block, so the number of slices used by other parts of the design is $2873 - 837 = 2036$. There are altogether 13300 slices, therefore $13300 - 2036 = 11264$ of them are available for more instances of `align()`, so we can conclude that $\left\lfloor \frac{11264}{837} \right\rfloor = 13$ align blocks could work in parallel on the same chip.

4.3. Future Work

4.3.1. Optimization of Parameters

After we have successfully laid the groundwork for our system, it now needs to be tested extensively, and its parameters must be refined as well. For lack of time, we could not apply it to a sizable database containing real-world data, i.e., backbone torsion angles from many actual proteins.

The consequence is that we were not able to tune the two currently-existing parameters (K and P), nor did we have the chance to evaluate other families of possible scoring functions besides the simple and naïve one we have devised. Finding the optimal parameters requires parameter sweeping, that is, exhaustive search of the — appropriately discretized and sampled — (K, P) space. This directly translates to hundreds, thousands or even millions of runs over a well-known, hopefully representative, thus likely large, dataset. It also needs to be decided which point in this space has the biologically optimal outcome, and this necessitates either manual work or automation; alas, the former is laborious, while the latter is difficult to implement reliably. Of course, parameter

Component	Used	Available	Utilization (%)
Slice LUTs	946	53200	1.78%
Slice Registers	1339	106400	1.26%
F7 Muxes	1	26600	<0.01%
Slice	421	13300	3.17%
LUT as Logic	866	53200	1.63%
LUT as Memory	80	17400	0.46%
LUT Flip Flop Pairs	1301	53200	2.45%
Block RAM Tile	2.5	140	1.79%
DSPs	0	220	0.00%

Table 4.9. Logic elements used by the biggest AXI DMA, `out_scores` (#3)

sweeping is not the only optimization method, but it is the only one that is relatively simple to supervise manually, and it also has the nice property that it finds the global optimum on its domain. There exist faster ways of optimization based on the derivative of the objective function, such as gradient descent and Newton's method, but these are only guaranteed to find a local extremum, not necessarily the global one.

On the other hand, experimenting with the scoring function implies frequent recompilation of the HLS code, a process that is relatively slow to iterate. However, it would be very insightful to try out a few other scoring schemes based on nearly universally used distance functions and see which one performs best. Two such examples are the Manhattan distance ($P = 1$ norm), which gives rise to the scoring function $s(\Phi_1, \Psi_1, \Phi_2, \Psi_2) := K - [d(\Phi_1, \Phi_2) + d(\Psi_1, \Psi_2)]$, and a family of exponentially-downweighted scoring functions, of the form $s(\Phi_1, \Psi_1, \Phi_2, \Psi_2) := e^{-C \cdot m(\Phi_1, \Phi_2, \Psi_1, \Psi_2)}$, where m is a metric in (Φ, Ψ) space and $C \in \mathbb{R}^+$.

Another improvement that could bring us closer to a biologically meaningful result is a different gap penalty. For example, affine, logarithmic and log-affine gap penalties have been used in the past; however, the logarithmic scheme has not proven very useful [30]. The affine gap penalty is a more widely accepted formula, but due to the relatively high complexity and resource requirements of its implementation, we have postponed its use for now.

Overall, these measures would be necessary for ensuring that the mathematical optimum computed by the algorithm coincides with the best spatial alignment or, preferably, with the most valuable genetic and evolutionary relationships. Properly and robustly checking this property requires cross-validation of the output on several sets of true biological data extracted from genuine proteins. Because of the shortness of time, this step has also been omitted.

4.3.2. Parallelization in More Dimensions

As it can be observed, a single instance of the processing element works somewhat slower than the software simulation on a relatively powerful, recent desktop computer. Consequently, in order to be practical and useful, it needs to be parallelized further. As described before, the next step in the direction of higher-level parallelization is the

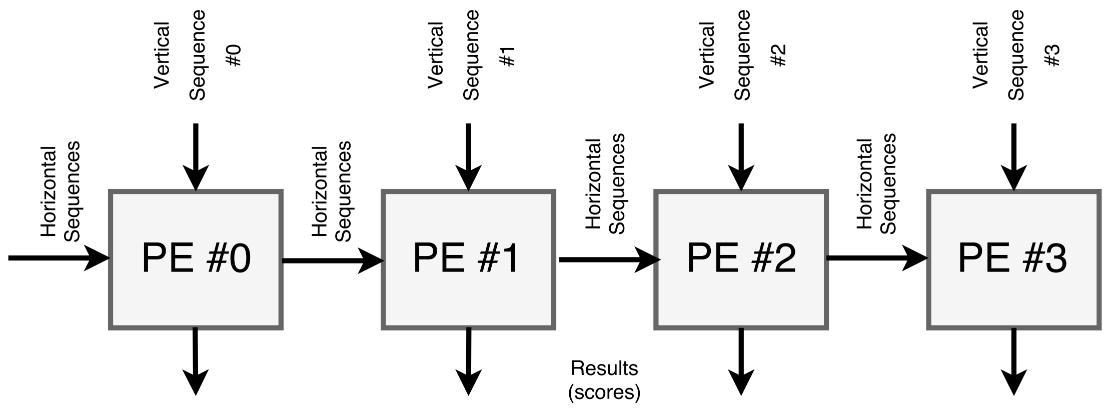

Figure 4.1. Processing elements in a linear chain topology

simultaneous processing of many pairs of proteins, as these computations are independent of one another. For this, we need to stamp out many copies of the HLS alignment block.

When the blocks are instantiated, sequence data needs to be distributed between them. There are two easy ways of doing that. The first — almost self-evident — approach is to create a fanout tree that repeats its input stream as many times as there are processing elements. The first processing unit then compares the first sequence in the stream to every other sequence, the second processor ignores the first sequence, and compares the second sequence to the third one and subsequent ones, the third unit ignores the first two sequences and starts alignment at the third one; and so forth. It is also possible that the fanout tree itself takes care of skipping the first n sequences when it provides data for the processing element at index n.

One could also prepare a tree-based architecture in a slightly different manner. In this lineup, the same vertical sequence is streamed to each processing element in one go (because it is only read once), and the fanout tree splits up the data corresponding to the horizontal sequences, so that every processing element gets a different subset thereof.

The second approach is to connect the processing elements in a chain topology, whereby each of them has an auxiliary output stream in addition to the one delivering the scores. This second output simply forwards the stream of horizontal sequences to the next element in the chain, while the vertical streams are different for each element. Intuitively, we would expect that this chain structure, as pictured in figure 4.1, allows for a higher clock frequency, as it can result in neighboring elements being physically close to each other, so less wiring is necessary for distributing sequence data between processors. The sequential nature of this layout also allows the synthesis tool to insert delay registers between elements when necessary, thereby reducing the number of operations that it tries to perform during the same clock cycle. This also decreases the complexity and the length of internal connections, which again results in a higher possible clock rate. Simpler wiring has the added benefit of yielding faster place and route, which is important because place and route is an NP-hard problem that takes up a big portion of HLS compilation time.

Regardless of the structure, we should pay attention to sorting the sequences by length. Whether the order is ascending or descending (i.e., whether shorter or longer structures are at the beginning) should not matter; the gist of sorting is that it minimizes the difference between the length of consecutive sequences, which is useful for efficiency reasons. When two processors work in parallel, and they get input periodically, and the time points when they start receiving their input are synchronized, then the one performing the shorter computation always stalls waiting for the other, slower one. So, by reducing the difference between input sizes, we can ensure that more time is spent usefully, in actual processing, and less time is wasted waiting for other components of the system.

A related optimization that can be applied to a chain topology is streaming longer inputs to processing elements located near the beginning of the chain, so that they can start computing as soon as possible, while the ones near the end of the chain get shorter sequences. This allows PEs to finish processing at about the same time, so the elements that started earlier need not wait for peers that only began working after a delay.

5. Summary and Conclusions

In this work, we have established an efficient parallel implementation of the Smith–Waterman algorithm for fast structural alignment of proteins on programmable logic devices. We have analyzed the resource consumption and the energy dissipation of the hardware, and we have relaxed certain requirements, such as the amount of operative memory needed, compared to previous work in the field.

With this approach, we have have laid the foundations of a well-scalable architectural design. Realization of further upscaling of the system and optimizing its parameters still remains an open question, however.

6. Acknowledgments

I would like to warmly thank my advisors, Zoltán Nagy and Zoltán Gáspári, for all their help, valuable professional advice, and kind patience.

I would also like to express my gratitude towards my entire family for their moral support and continued encouragement during my studies.

Bibliography

[1] L. Brocchieri and S. Karlin, “Protein length in eukaryotic and prokaryotic proteomes”, *Nucleic acids research*, vol. 33, no. 10, p. 3390, 2005.

[2] Y. Zhuang, F. Ma, J. Li-Ling, X. Xu, and Y. Li, “Comparative analysis of amino acid usage and protein length distribution between alternatively and non-alternatively spliced genes across six eukaryotic genomes”, *Molecular biology and evolution*, vol. 20, no. 12, pp. 1978–1985, Jun. 2003. DOI: `10.1093/molbev/msg203`.

[3] O. Carugo and S. Pongor, “Protein fold similarity estimated by a probabilistic approach based on C_α-C_α distance comparison”, *Journal of molecular biology*, vol. 315, pp. 887–898, 2002.

[4] Z. Gáspári, K. Vlahoviček, and S. Pongor, “Efficient recognition of folds in protein 3-d structures by the improved PRIDE algorithm”, *Bioinformatics*, vol. 21, pp. 3322–3323, 2005.

[5] R. H. Lathrop, “The protein threading problem with sequence amino acid interaction preferences is NP-complete”, *Protein engineering*, vol. 7, no. 9, pp. 1059–1068, 1994.

[6] C. A. Orengo, A. D. Michie, S. Jones, D. T. Jones, M. B. Swindells, and J. M. Thornton, “CATH — a hierarchic classification of protein domain structures”, *Structure*, vol. 5, no. 8, pp. 1093–1108, 1997. DOI: `10.1016/S0969-2126(97)00260-8`.

[7] I. Sillitoe, T. E. Lewis, A. L. Cuff, S. Das, P. Ashford, N. L. Dawson, N. Furnham, R. A. Laskowski, D. Lee, J. G. Lees, S. Lehtinen, R. A. Studer, J. M. Thornton, and C. A. Orengo, “CATH: Comprehensive structural and functional annotations for genome sequences”, *Nucleic acids research*, vol. 43, Jan. 2015. DOI: `10.1093/nar/gku947`.

[8] W. R. Taylor and C. A. Orengo, “Protein structure alignment”, *Journal of molecular biology*, vol. 208, no. 1, pp. 1–22, 1989. DOI: `10.1016/0022-2836(89)90084-3`.

[9] C. A. Orengo and W. R. Taylor, “SSAP: Sequential structure alignment program for protein structure comparison”, *Methods in enzymology*, vol. 266, pp. 617–635, 1996. DOI: `10.1016/s0076-6879(96)66038-8`.

[10] N. V. Grishin, “Fold change in evolution of protein structures”, *Journal of structural biology*, vol. 134, pp. 167–185, 2001.

[11] L. Holm and C. Sander, “Protein structure comparison by alignment of distance matrices”, *Journal of molecular biology*, vol. 233, pp. 123–138, 1993.

[12] P. K. Agarwal, N. H. Mustafa, and Y. Wang, “Fast molecular shape matching using contact maps”, *Journal of computational biology*, vol. 14, no. 2, pp. 131–143, 2007. DOI: `10.1089/cmb.2007.0004`.

[13] N. Malod-Dognin, “Protein structure comparison: From contact map overlap maximisation to distance-based alignment search tool. modeling and simulation”, PhD thesis, Université Rennes, 2010.

[14] G. Lancia, R. Carr, and B. Walenz, “101 optimal PDB structure alignments: A branch-and-cut algorithm for the maximum contact map overlap problem”, in *RECOMB*, 2001, pp. 193–202.

[15] L. Holm, S. Kääriäinen, P. Rosenström, and A. Schenkel, “Searching protein structure databases with DaliLite v.3.”, *Bioinformatics*, vol. 24, no. 23, pp. 2780–2781, Dec. 2008. DOI: `10.1093/bioinformatics/btn507`.

[16] P. E. Bourne and H. Weissig, *Structural bioinformatics*. Wiley-Liss, Inc., 2003.

[17] Z. Gáspári, “Structure validation”, University Lecture, 2016, [Online]. Available: `https://wiki.itk.ppke.hu/twiki/pub/PPKE/StructBioinf/StructureValidation.pdf`.

[18] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences”, *Journal of molecular biology*, vol. 147, pp. 195–197, 1981.

[19] 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*, vol. 48, no. 3, pp. 443–453, 1970.

[20] A. G. Murzin, S. Brenner, T. Hubbard, and C. Chothia, “SCOP: A structural classification of proteins database for the investigation of sequences and structures”, *Journal of molecular biology*, vol. 247, no. 4, pp. 536–540, 1995. DOI: `10.1016/S0022-2836(05)80134-2`.

[21] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation”, in *ACM SIGPLAN 2007 conference on programming language design and implementation (PLDI 2007)*, 2007. DOI: `10.1145/1273442.1250746`. [Online]. Available: `http://valgrind.org/docs/valgrind2007.pdf`.

[22] “Programming languages — C++”, International Organization for Standardization, Working Draft, Standard, May 2013. [Online]. Available: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf`.

[23] *Vivado design suite user guide — high-level synthesis*, English, version v2014.1, Xilinx, Inc., May 30, 2014, 660 pp., 2016-03-26.

[24] *ZedBoard hardware user’s guide*, English, version 1.1, Avnet, Inc. and Digilent, Inc., Aug. 1, 2012, 37 pp., 2012-08-02.

[25] P. Zhang, G. Tan, and G. R. Gao, "Implementation of the Smith–Waterman algorithm on a reconfigurable supercomputing platform", in *HPRCTA '07 proceedings of the 1st international workshop on high-performance reconfigurable computing technology and applications: Held in conjunction with SC07*, Altera Corporation, Nov. 11, 2007. DOI: 10.1145/1328554.1328565. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01035.pdf.

[26] B. T. Szabó, "Fehérjeszekvenciák szekvenciaillesztésének gyorsítása programozható logikai áramkörök segítségével", Bachelor's thesis, PPKE-ITK, 2013.

[27] D. A. Tálas, "Torziósszög-párok távolságán alapuló fehérjeszerkezet-összehasonlítási módszer fejlesztése és alkalmazása FPGA-n", Bachelor's thesis, PPKE-ITK, 2015.

[28] *Xilffs_polled_example.c.* [Online]. Available: https://goo.gl/VkZU6l.

[29] *Battery Health*, version 5.4, FIPLAB, Dec. 1, 2016. [Online]. Available: https://itunes.apple.com/hu/app/battery-health-monitor-battery/id490192174.

[30] R. A. Cartwright, "Logarithmic gap costs decrease alignment accuracy", *BMC Bioinformatics*, vol. 7, no. 527, Dec. 2006. DOI: 10.1186/1471-2105-7-527. [Online]. Available: http://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-7-527.

Appendix A.

List of annexes

The archive `https://users.itk.ppke.hu/~gorar/BScThesis.zip` and the attached DVD contains:

- LaTeX sources, supplementary files, and a compiled PDF in the `LaTeX` directory;
- the C++ source code for the HLS block and its driver for simulation in the `FPGA` directory;
- the C source code for the bare-metal application in the `ARM` directory;
- the C and Python source of the testing framework (`smith_waterman.py` and `multi_gen_random_seqs.c`) in the `FPGA/test` directory;
- a set of sample inputs (`INPUT.BIN`) and the corresponding scores (`OUTPUT.BIN`) for verification in the same directory;
- the project files for the Vivado, Vivado HLS and Xilinx SDK integrated development environments, preserving the original directory structure, in the `Projects` directory.