Technical University of Crete, Greece
School of Electrical and Computer Engineering

An HLS Approach to Accelerating the Needleman-Wunsch Dynamic Programming Algorithm

Evangelos Mageiropoulos

Thesis Committee
Professor Dionisios Pnevmatikatos (ECE) (Supervisor)
Professor Apostolos Dollas (ECE)
Associate Professor Ioannis Papaefstathiou (ECE)

Chania, March 2017

Abstract

In the field of hardware-based parallelism, Field Programmable Gate Arrays (FPGAs) play a key role as a platform to provide a high level of parallelization. In recent years, High Level Synthesis (HLS) software has been introduced as a tool for developing on FPGAs. This software brings a higher level of abstraction compared to traditional programming methods that are used to directly describe the Register-Transfer Level (RTL) design to be implemented on an FPGA. Xilinx, the current largest FPGA vendor has recently introduced Vivado HLS, as a part of the Vivado design suite.

In this thesis, we present our approach at implementing the Needleman-Wunsch dynamic programming algorithm for protein sequence alignment on Vivado HLS, with the main goal to achieve the highest level of parallelism and acceleration. This study contributes an analysis of the algorithm under the scope of parallelism, its data dependencies and the limitations they set, as well as data locality, for which we developed a cache simulator in Java. Additionally, it provides an analysis of Vivado HLS regarding its performance on accelerating the algorithm and presents a comparison of the methods offered by the tool for optimizing the implementation.

Initially, we briefly present the Needleman-Wunsch algorithm and describe its function. Subsequently, we analyze it in regard to the maximum level of parallelism that can be achieved in respect to the limitations of the algorithm and proceed to further examine the available methods of parsing its dynamic programming matrix under the scope of data locality. Here we present the cache simulator, the results of which are important for the comparison of the efficiency of the aforementioned methods regarding their hit/miss rates when accessing the cache memory. Afterwards, we present two discrete implementations of the algorithm on Vivado HLS: One using the code of a software implementation of the algorithm and one with code we developed considering the results of the analysis of the algorithm. In the former case, we used a set of directives offered by Vivado HLS, in order to optimize the performance of our implementation; without however achieving parallelism on the design. In the latter one, which describes a parallel implementation of the algorithm, we observed the inability of Vivado HLS to achieve the expected acceleration of the design. Finally, we present the conclusions of our study, which provide input for further work on optimizing the implementation of the algorithm and comparing it with alternative implementations.

Acknowledgements

I would first like to thank my thesis advisor, Dr. Dionysios Pnevmatikatos, for helping me shape my interest towards computer hardware engineering through his classes and subsequently accepting me for this thesis and mentoring me through its course. I would also like to express my gratitude to Dr. Apostolos Dollas and Dr. Yannis Papaefstathiou for their interest in my work and for contributing to its evaluation as members of the thesis committee.

Thanks are also due to Dr. Gregory Chrysos and George Charitopoulos for their valuable assistance on this thesis. They gladly provided input on any issues that occurred and their support was always present, especially when I needed it the most.

Finally, a wholeheartedly thank you to my family and my sweet Andriani for their love, support and encouragement throughout my studies and through the course of my thesis. This accomplishment would not have been possible without them.

Contents

List of Figures

LIST OF FIGURES

Chapter 1

Introduction

2001 was the year the first general-purpose processor featuring multiple cores on its die, the POWER4 processor by IBM, was released to the public [1]. In 2005, Intel followed suit, introducing their first lineup of multicore CPUs and since then, parallel architectures have become the norm. This turn from a single core to a multicore (and even *manycore*) environment in high availability allowed for true application parallelization with great performance benefits and speedups. However, the switch from sequential to parallel computing makes programming harder, more demanding and at the risk of not achieving the desired improvement in performance. Therefore, the need for the categorization of the algorithmic motifs that are considered to be important in regard to the parallel landscape has surfaced. Researchers at Berkeley introduced the *Dwarfs of High-Performance Computing*, a collection of algorithmic methods that "capture a pattern of computation and communication" [3] to cover this need.

The parallel landscape is also defined by the available hardware platforms that support parallelization, a field where FPGAs play a key role. Their ability to be programmed at a hardware level gives them a clear advantage on the level of parallelism they can achieve when compared to general purpose CPUs. Although the main method of describing an algorithm for implementation on an FPGA is by using a hardware description language (VHDL, Verilog), in recent years, there has been an attempt to provide a high level of abstraction in the design workflow with the introduction of High Level Synthesis software. HLS tools allow for defining a design using a high-level programming language and exporting an RTL design defined in a hardware description language. Xilinx, the current largest FPGA manufacturer provides the Vivado High Level Synthesis software as a part of the Vivado Design Suite, a software suite for developing on Xilinx FPGAs.

High Level Synthesis tools are designed to provide ease of use and fast development times. However, it is important to be evaluated on the performance and efficiency of their resulting RTL design. The need for evaluation is especially prominent in the field of parallel algorithmic motifs, where FPGAs are expected to have a significant performance advantage over CPU/software solutions.

In this study, we present the implementation of the Needleman-Wunsch algorithm for sequence alignment that follows the dynamic programming algorithmic motif included in the Dwarfs, on Vivado HLS. The study contributes an analysis of the algorithm under the scope of parallelism, its data dependencies and the limitations they set, as well as data locality, for which we developed a cache simulator in Java. Furthermore, it provides an analysis of Vivado HLS as a tool to accelerate the algorithm enhance its performance and presents a comparison of the methods it offers towards optimizing the implementation. In particular:

- In **Chapter 3** we present the Needleman-Wunsch algorithm as a method of aligning a pair of protein sequences and the steps it follows towards finding the optimal global alignment.
- In **Chapter 4** we provide an in-depth analysis of the Needleman-Wunsch algorithm under the scope of parallelization and demonstrate the two possible methods of parsing the dynamic programming matrix. We also introduce the cache simulator we developed in order to determine the cache hit and miss rates of those methods and discuss the results.
- In **Chapter 5** we present two discrete implementations of the algorithm on Vivado HLS; one utilizing a preexisting code targeting a software implementation and one developed considering the results we collected in Chapter 4. We then attempt to enhance the performance of the implementations with the use of optimization directives provided by Vivado HLS and present the results.
- In **Chapter 6** we conclude our study and discuss our observations. Additionally, we proceed to suggest topics for future work to further expand our study.

Chapter 2

Related Work

2.1 The Dwarfs of High-Performance Computing

A dwarf is defined as an "algorithmic method that captures a pattern of computation and communication" [3]. The dwarfs were initially presented by Philip Colella in 2004 [2] as a set of seven algorithmic motifs that he believed would be important for physical sciences and engineering for at least the next decade. Colella's work was further expanded by researchers at the University of California, Berkeley [3], under the consideration of the changing landscape of hardware-level parallelization. Their research aimed at extending the algorithmic fields represented by the dwarfs to cover a larger range of applications; the dwarfs were subsequently expanded to the following 13 algorithmic domains:

- **Structured Grids**: A pattern represented by a regular multidimensional grid, where points on it are conceptually updated together - either in place or between two versions of the grid. This method has a high temporal locality, as all points are updated using data from a small neighborhood around each one. The grid can be split into smaller grids that contain areas of interest; transitions between those grids may happen dynamically.

- **Unstructured Grids**: Data are represented in an irregular grid, where their locations are selected usually by the characteristics of the application. As in **Structured Grids**, points are conceptually updated together. Because the update to any point requires determining a list of neighboring points and loading values from them, there are multiple levels of memory reference indirection.

- **Spectral Methods**: Data refers to the frequency spectrum. There is a combination of multiply-add operations and a specific pattern of data permutation. In some stages of the calculation, an all-to-all communication might happen, while there might be no communication in others.

- **Dense Linear Algebra**: Data are stored in dense matrices or vectors and there are three levels of operations - Level 1 (vector/vector), Level 2 (matrix/vector) and Level 3 (matrix/matrix). Most applications use unit-stride memory accesses to read row data and stride accesses to read column data.

- **Sparse Linear Algebra**: Data used for calculations stored in matrices and include many zero values - therefore the matrices are usually compressed to reduce storage and bandwidth demands when accessing all nonzero values. Due to the data being compressed, access is performed with indexed loads and stores.

- **N-Body Methods**: Computations depend on interactions between many points on the grid. Variations include particle-particle methods, where every point depends on all others, leading to complexity of $O(n^2)$ and hierarchical particle methods that combine forces from multiple points to reduce the computational complexity to $O(n \log n)$ or $O(n)$.

- **MapReduce**: An algorithmic motif which relies on the implementation of the functions `Map()` and `Reduce()` to assign workload in a multi-processor environment. A series of repeated random runs produces the result set. Due to the fact that it can be very easily parallelized, it is considered *embarassingly parallel*.

- **Combinational Logic**: A method that performs simple operations on a large set of data, e.g. operations on Cyclic Redundancy Codes.

- **Graph Traversal**: The algorithm traverses a set of objects in a graph, while examining their characteristics.

- **Dynamic Programming**: The problem is split into smaller and simpler sub-problems that are evaluated towards finding a solution to the initial problem.

- **Backtrack and Branch-and-Bound**: An algorithmic method searching a large space towards finding a globally optimal solution. It is used on implementations search and global optimization problems.

- **Graphical Models**: Sets of graphs with nodes and edges, where each node represents a variable and each edge a conditional probability for transitioning to a node.

- **Finite State Machines**: A non-empty set of states and functions describing transitions between the states are used to model a computation. In some cases, the machine can be split into smaller machines that can run in parallel.

2.2 Vivado High Level Synthesis

The Vivado High Level Synthesis is a tool distributed by the major FPGA manufacturer Xilinx as a part of the Vivado development suite, that transforms a programming algorithm defined in **C**, **C++**, **SystemC** or as an **OpenCL** API C kernel into a Register Transfer Level (**RTL**) implementation, ready to be synthesized on a compatible Xilinx FPGA. Vivado HLS introduces a higher level of abstraction to the designer, allowing them to describe an algorithm in a general-purpose programming language rather than a hardware description language (VHDL, Verilog) in an attempt to improve productivity and enhance system performance. [14]

With Vivado HLS, the designer can describe the algorithm using any of the compatible C-based languages [1]. A function that acts as a main entry for the design must be set as a *top-level function*; its arguments will be implemented as an interface on the RTL when synthesized. After providing the code, the designer can validate its correctness against a known good output, analyze its performance using the bundled inspection tools, transform the design to RTL and export it to an RTL package in a desired IP format. This package can subsequently be imported and used as a core in other Vivado designs.

Vivado HLS gives the designer the opportunity to specify how parts of the source code should be transformed in the synthesis process through the use of **directives**. Those are a set of `#pragma` statements that can be used by the designer to instruct the Vivado HLS on how it should transform the code area the directive refers to, with the aim of achieving a more refined and optimized result. The main code areas for which directives can be applied are:

- The **interfaces** that will be implemented on the arguments of the *top-level function*, e.g. AXI4, bus, etc.

[1] Vivado HLS only supports a limited subset of the operations each programming language provides. For example, there are limitations on the implementation of pointers and no provision for dynamic memory allocation.

- The **storage elements**, in particular **arrays**: The designer can select how arrays are implemented into block rams and registers.
- The **loops** that appear in a function body. Those can be unrolled, pipelined etc. in order to enhance parallelism.

In section 7.1 of the Appendix, we present a detailed list of the directives available on Vivado HLS.

Due to HLS becoming widely available only recently, it is necessary to evaluate it under different algorithmic motifs and assess its performance and any benefits in comparison to a traditional RTL design methodology. Therefore, there has been research that focuses on providing an evaluation of the methods the tool provides for enhancing the performance of the design [7] and compare the results with direct, RTL implementations [8]. The results of those studies indicates that Vivado HLS can introduce speedups on the design and perform more efficiently latency and resource-wise.

Additionally, in order to further provide a layer of abstraction, a *dynamic memory management* system has been introduced for Vivado HLS, since it does not natively support this feature[1] [9, chapter 5]. This makes easier sharing memory between multiple processing elements inside the design with varying memory demands throughout its runtime.

2.3 Work on Dwarfs

The collection of Dwarfs as a set of well-defined algorithmic methods has been implemented in OpenCL in the OpenDwarfs project [4]. OpenCL (Open Computing Language) is a programming framework that allows for platform-agnostic code to be executed on heterogeneous platforms that usually include CPUs, GPUs and, more recently, FPGAs. The study in [6] raises the concern that an architecture-agnostic approach would cause an inefficient implementation that will not utilize the resources of an FPGA optimally and proceeds to compare unoptimized and optimized versions of algorithms following the *structured grids* and *N-body* methodologies as described in the Dwarfs.

The study in [5] further expands [6] and introduces the Needleman-Wunsch algorithm as an implementation of the *dynamic programming* algorithmic method from the Dwarfs. It then proceeds to implement its kernel on OpenCL and execute it on the Opteron 6272 CPU, as well as the HD7660D and HD6550D GPUs.

An implementation of the Needleman-Wunsch algorithm on HLS is presented in [10]. In this paper, the researchers used the CoDeveloper HLS suite that accepts code in Impulse C and generates HDL code, in a fashion similar to Vivado HLS. They follow a different approach from us however, as they directly proceed to implement a multi-process system with streaming input and output from and to a CPU-controlled storage server. Additionally, they do not perform a data locality analysis for efficient cache usage. They conclude that the HLS design is faster than an HDL implementation; however the need to describe hardware still stands (even with a high level language).

Chapter 3

The Needleman-Wunsch Dynamic Programming Algorithm

3.1 Introduction

The Needleman-Wunsch algorithm was developed by Saul Needleman and Christian Wunsch [11]. It is used in bioinformatics to calculate the *alignment* of a pair of *sequences* (collections of nucleotides or amino acid residues), i.e. a way of arranging the sequences in order to find regions of similarity between them. The Needleman-Wunsch algorithm falls under the category of *global alignment* algorithms, which carry the alignment throughout whole pair of similarly sized sequences, in contrast to *local alignment* algorithms, such as the Smith-Waterman, which find particular regions of similarity on pairs of dissimilar sequences.

The algorithm follows the dynamic programming methodology to perform the global alignment, as it splits the original problem into small individual subproblems and solves them towards finding the solution for the original one. For this procedure, it utilizes a two-dimensional *dynamic programming matrix* to store the solutions of the subproblems. For a pair of sequences with lengths m, n, the matrix has size $(m+1)$ x $(n+1)$, which causes the algorithm to have a time as well as space complexity of $O(mn)$.

3.2 Algorithm Presentation

On its simplest form, the Needleman-Wunsch algorithm uses the scoring function $S(x_i, y_j)$ which accepts a pair of amino acids or proteins and returns their match/mismatch score. A simple scoring function might simply return a given value if the input pair matches or not. In a

more complicated scoring system however, a similarity matrix might be used instead - e.g. the BLOSUM62 similarity matrix which is presented in section 7.3 of the Appendix, which contains different scores for each pair of inputs. Additionally, it considers a gap penalty value, $v_{gap\ open}$ to account for opening gaps while aligning the sequences. The algorithm first initializes the first row and column of the dynamic programming matrix, v_{dp}, using the formulas:

$$v_{dp}(0,0) = 0 \tag{3.1}$$

$$v_{dp}(i,0) = v_{dp}(i-1,0) + v_{gap\ open}, i > 0 \tag{3.2}$$

$$v_{dp}(0,j) = v_{dp}(0,j-1) + v_{gap\ open}, j > 0 \tag{3.3}$$

It then uses the following function to fill the matrix, beginning from $v_{dp}(1,1)$:

$$v_{dp}(i,j) = \max \begin{cases} v_{dp}(i-1,j-1) + S(x_i, y_j) \\ v_{dp}(i-1,j) + v_{gap\ open} \\ v_{dp}(i,j-1) + v_{gap\ open} \end{cases} \tag{3.4}$$

For each $v_{dp}(i,j)$ calculated, the algorithm keeps the cell that was selected from the max function and creates a path to cell $(0,0)$. When the calculation of the dynamic programming matrix reaches the bottom right cell and is completed, it traces back the path to $(0,0)$. This procedure creates the sequence alignment, in the sense that diagonal movement matches pairs on both sequences, horizontal movement introduces a gap in the sequence mapped to the rows of the matrix and horizontal movement introduces a gap in the sequence mapped to the columns of the matrix[1].

3.3 Example of Usage

Consider the sequences `ACGCATCA` and `ACTGATTCA` to be aligned with the Needleman-Wunsch algorithm, with a match score of $v_{match} = 2$, a mismatch penalty of $v_{mismatch} = -3$ and a gap penalty $v_{gap\ open} = -2$ [12]. As mentioned, the algorithm initializes a two-dimensional matrix with the size of $(m+1)$ x $(n+1)$ given sequence lengths m and n, respectively; in this case, the matrix has a size of 9x10.

[1]Note that there might be more than one possible global alignments for a pair of sequences. The Needleman-Wunsch algorithm selects one of them.

		A	C	T	G	A	T	T	C	A
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
A	-2									
C	-4									
G	-6									
C	-8									
A	-10									
T	-12									
C	-14									
A	-16									

Figure 3.1: Initialized 9x10 dynamic programming matrix

The algorithm proceeds to initialize the first row and column of the matrix using the equations 3.1-3.3. The resulting initialized matrix appears in figure 3.1.

Following the matrix initialization, the algorithm begins to calculate the value of each cell, starting from cell $(1, 1)$, using the equation 3.4. In the case of the cell $(1, 1)$, its value is calculated as:

$$v_{dp}(1,1) = \max \begin{cases} v_{dp}(0,0) + v_{match} \\ v_{dp}(0,1) + v_{gap\ open} \\ v_{dp}(1,0) + v_{gap\ open} \end{cases} = \max \begin{cases} 0+2 \\ -2-2 \\ -2-2 \end{cases} = 2$$

Additionally, the algorithm keeps track of the cell that was selected from the max function - in this case, cell $(0, 0)$. Thus, the dynamic programming matrix is updated as shown in figure 3.2.

		A	C	T	G	A	T	T	C	A
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
A	-2	2								
C	-4									
G	-6									
C	-8									
A	-10									
T	-12									
C	-14									
A	-16									

Figure 3.2: The dynamic programming matrix with cell $(1, 1)$ calculated

The algorithm subsequently proceeds to calculate the values of the remaining cells in the

dynamic programming matrix. Figure 3.3 presents the resulting matrix. It then traces the path from the bottom right cell back to cell $(0, 0)$, resulting in the following alignment:

```
ACTG-ATTCA
||  |  ||  ||
AC-GCAT-CA
```

		A	C	T	G	A	T	T	C	A
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
A	-2	2	0	-2	-4	-6	-8	-10	-12	-14
C	-4	0	4	2	0	-2	-4	-6	-8	-10
G	-6	-2	2	1	4	2	0	-2	-4	-6
C	-8	-4	0	-1	2	1	-1	-3	0	-2
A	-10	-6	-2	-3	0	4	2	0	-2	2
T	-12	-8	-4	0	-2	2	6	4	2	0
C	-14	-10	-6	-2	-4	0	4	2	6	4
A	-16	-12	-8	-4	-5	-2	2	1	4	8

Figure 3.3: The completed dynamic programming matrix with the traceback path [12]

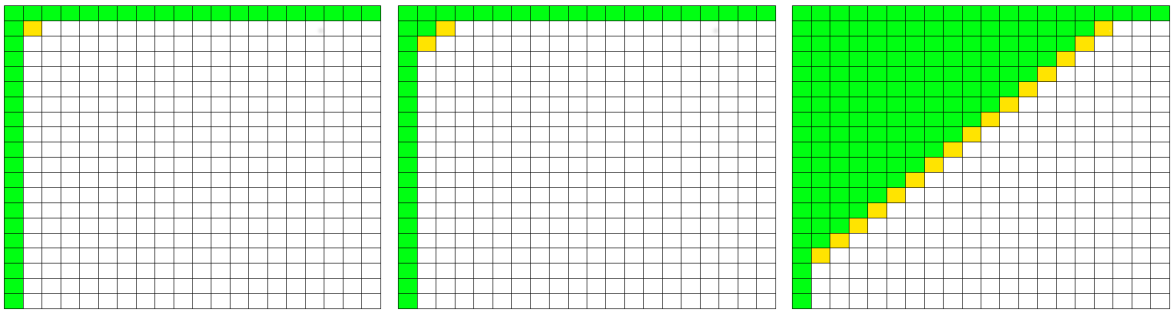
Chapter 4

Analysis of the Algorithm

The main advantage of an algorithm implementation on an FPGA is the high level of parallelism that can be achieved in contrast to a traditional CPU/software implementation. Therefore, it is important that the algorithm is analyzed for its maximum level of parallelism, while considering any delays that might occur due to data transfers and might hinder performance and introduce latency on the design. In this chapter, we analyze the Needleman-Wunsch algorithm under the scope of data dependencies and we subsequently examine the efficiency of the findings regarding data locality. Finally, we suggest an implementation of the design, considering the findings of this analysis.

4.1 Stencil Computation

In scientific computing, a **stencil** defines the computation of an element in an n-dimensional spatial array at time t as a function of neighboring array elements at time $t-1$,...,$t-k$. The n-dimensional array plus the time dimension span an $(n+1)$-dimensional spacetime. A **stencil computation** is a traversal of spacetime in an order that respects the data dependencies imposed by the stencil. Such computational method might impose delays when, on the memory hierarchy of the system performing the computation, the storage required to accommodate for data dependencies exceeds the size of the available cache, thus resulting in cache misses [13].



(a) Initial state (b) Step 1 (c) Step 15

Figure 4.1: Parsing steps on a reverse-diagonal matter: Green indicates cells that have been computed, yellow indicates cells that have their data dependencies satisfied but are yet to be computed and white indicates cells that do not have their data dependencies satisfied and thus cannot be computed at the current state.

4.2 Data Dependency Analysis

4.2.1 The Needleman-Wunsch Algorithm as a Stencil Array

By the algorithm's definition, each cell (i, j) in the dynamic programming matrix depends on the cells $(i-1, j-1)$, $(i, j-1)$ and $(i-1, j)$. This makes the matrix parsing scheme fall under the category of a **stencil computation**. Furthermore, it limits the possible level of parallelism to parsing the matrix in a reverse-diagonal manner, as shown in figure 4.1 of a sample 20x20 matrix.

When the matrix filling operation begins, only the value of a single cell can be calculated. Following its calculation, the number of cells ready for calculation rises to two. After each **timestep**[1], the number of cells available for calculation rises by 1, to a maximum of the size of the diagonal of the array - then diminishes by 1 until the end of the calculation. Parallelism-wise, the best possible solution would be to implement as many execution units to calculate the value of each cell, i.e. *workers*, as there are cells on the diagonal of the matrix, in order to parse it as fast as possible. Such solution however would be unfeasible when considering the limitations of available hardware when working on large matrices. Additionally, it would be a waste of resources, as the workers would be fully utilized for only a fraction of the process - in the case of a square matrix, only once, when calculating the reverse diagonal. Therefore,

[1] In a stencil computation, a timestep refers to the calculation of the cells in the stencil array that have their dependencies satisfied at a given time.

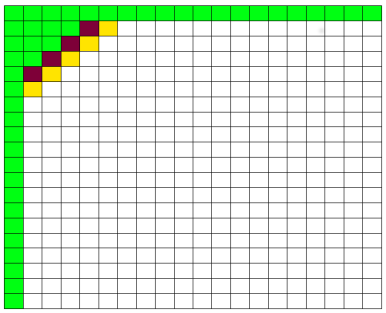


Figure 4.2: Array with 4 workers, step 4: This is the fourth step of the parsing and the first where all workers are utilized. Green indicates cells that have their values already calculated, red indicates the last cells that had their values calculated by the workers, yellow indicates the cells that have their dependencies satisfied and can be calculated and white indicates cells that do not have their dependencies satisfied. This step is common in both the diagonal and vertical parsing methods.

an implementation where the number of workers is smaller than the number of cells on the diagonal of the matrix would be expected. Under this premise, we proceed to present the possible methods for parsing the matrix, i.e. assigning matrix cells to workers for calculation.

4.2.2 Matrix Parsing Methods

There are two possible methods of scheduling the workload to the worker units with regard to how the matrix is being parsed: One following a diagonal parsing order and one following a vertical one, as shown in figures 4.2, 4.3, 4.4 and 4.5, in an example of a 20x20 matrix with 4 workers. A horizontal parsing method is also applicable - however it behaves in the exact way as the vertical one and thus is ignored in this study.

Regarding parallelism, the diagonal parsing method holds an advantage over the vertical one, as it adheres to the reverse-diagonal method which ensures that the maximum number of workers possible is always utilized, as stated earlier, leading to parse completion in the least amount of steps. In contrast, when $(Y-1) mod n_{workers} \neq 0$ (where Y is the number of matrix columns and $n_{workers}$ is the number of workers), there will be $n_{workers} - (Y-1) mod n_{workers}$ idle workers when working in the last set of columns with the vertical parsing method, as shown in figure 4.6.

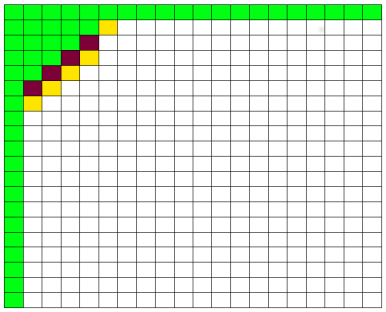
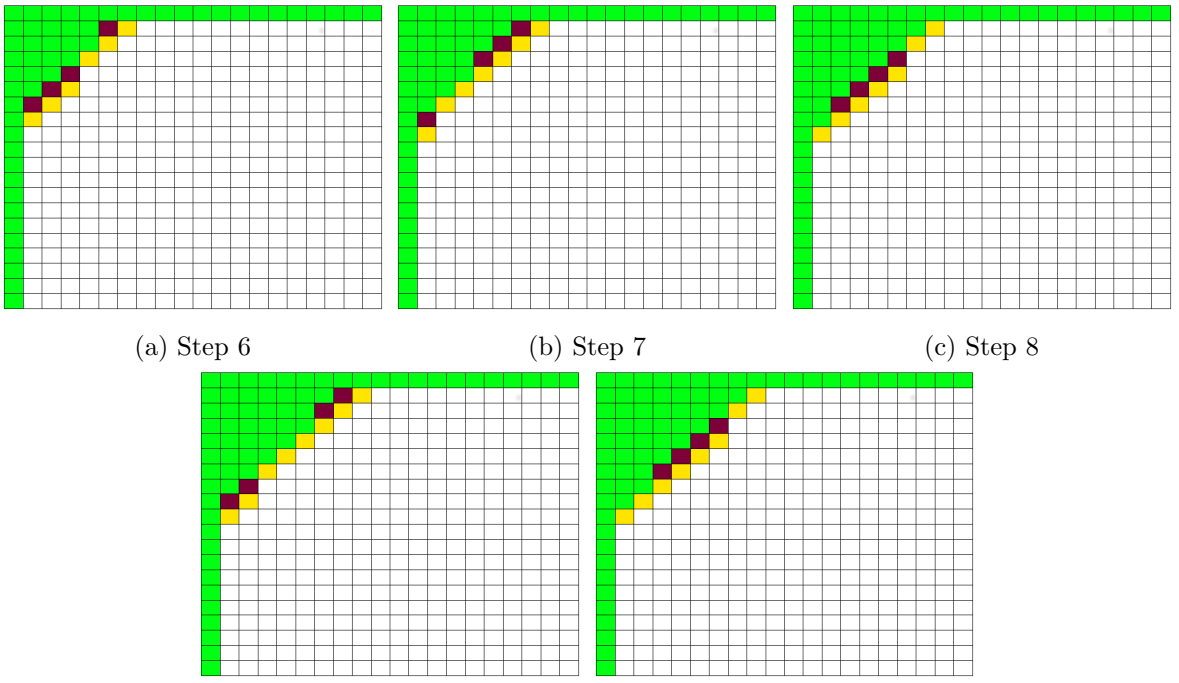


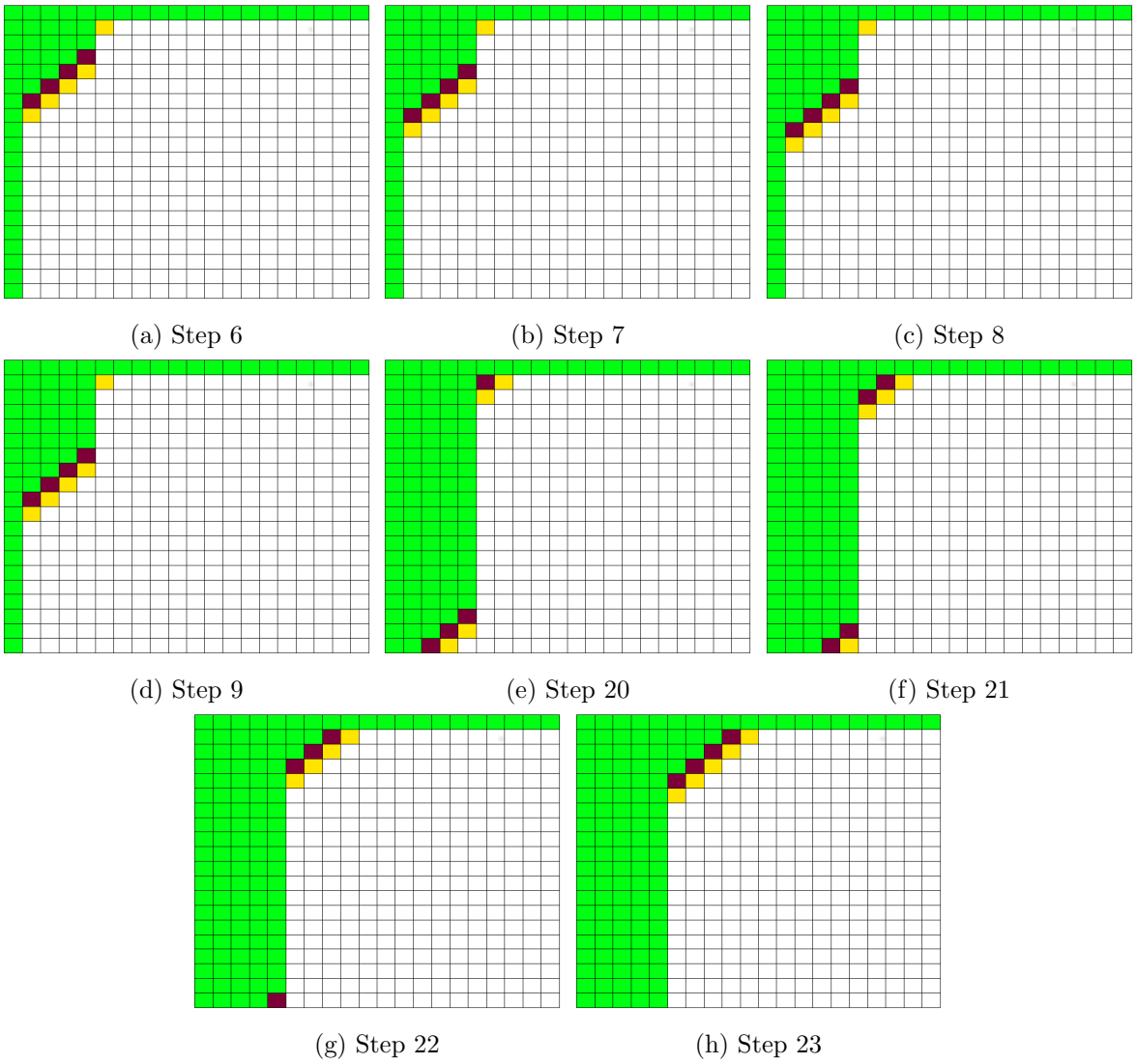
Figure 4.3: Array with 4 workers, step 5: This is the fifth step of the parsing and is common in both the diagonal with left anchor (i.e. diagonal parsing beginning on the left side of each diagonal) and vertical parsing methods.



(a) Step 6 (b) Step 7 (c) Step 8

(d) Step 9 (e) Step 10

Figure 4.4: Steps in a diagonal parsing method: The sixth to tenth step in the diagonal with left anchor parsing method. Workload is assigned in order to calculate cell values on current reverse diagonal and should workers be left non utilized, they proceed to the next reverse diagonal.

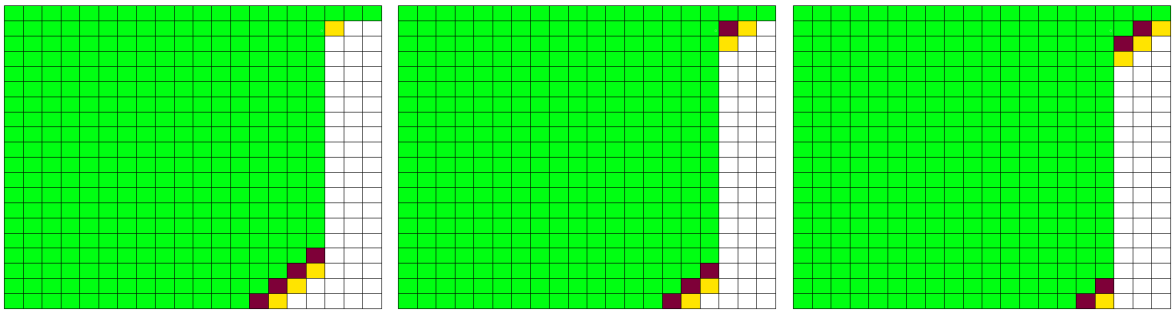


(a) Step 6 (b) Step 7 (c) Step 8

(d) Step 9 (e) Step 20 (f) Step 21

(g) Step 22 (h) Step 23

Figure 4.5: Steps in a vertical parsing method: The sixth to ninth and twentieth to twenty-third step in the vertical parsing method. Workload is assigned in vertical order. The twentieth to twenty-third steps show the procedure of changing the set of columns that are being parsed while keeping all the workers utilized.



(a) Step 76 (b) Step 77 (c) Step 78

(d) Step 79 (e) Step 80

Figure 4.6: Final steps in the vertical parsing method: In the case of 4 workers on a 20-by-20 matrix, there is one worker idle when parsing the last set of columns.

4.2.3 Conclusion

The analysis of the Needleman-Wunsch stencil array signifies that the parsing method to achieve the highest possible level of parallelism would be the diagonal one. However, the issue of data locality and efficiency regarding data transfers still needs to be examined and addressed. In the following section we compare the two parsing methods under the scope of data locality in order to minimize any slowdowns that might be caused from data that need to be retrieved from storage outside the FPGA.

4.3 Data Locality Analysis

4.3.1 Purpose

In any data processing design, an implementation of a cache memory to store data for fast access is necessary to improve data locality and performance. However, as stated earlier, a stencil computation might introduce delays during runtime, when the storage required to keep the dependencies needed for a timestep exceeds the available cache of the system. In such case, a cache miss will occur in order for the system to retrieve the data required and solve the dependency, which will cause the procedure to stall. Therefore, we need to analyze the parsing methods presented in the previous section under the scope of data locality and evaluate them on their efficiency regarding data reuse.

4.3.2 Determining Hits and Misses

In order to analyze their efficiency regarding cache hits and misses, each method needs to be tested with varying parameter values, for them to be studied under different use case scenarios. Those parameters are:

- The the dynamic programming matrix **size** (rows and columns), as defined by the length of inputs
- The **number of workers**
- The **cache size**

For simplicity's sake, during the analysis, we considered a **fully associative** cache to be present. This would achieve maximum cache utilization, under a best-case scenario for the

evaluation of cache hits and misses. Each cache entry consists of a pair of matrix row and column coordinates acting as a *tag* and the value of the corresponding cell for the **data** part.

In order to automate the procedure of measuring cache hits and misses, we developed a cache simulator in Java. The simulator accepts the aforementioned parameters as arguments and prints the number of hits and misses for each step, as well as their total amount up to that step. The results collected from the simulator execution were grouped into graphs for a better understanding of the performance of each method.

The simulator accepts the size of the dynamic programming matrix, the number of workers, the cache size and the desired parsing methods as arguments. It then initializes the following main elements:

- A **matrix** to hold the status of each cell - whether its value has been calculated, it has its dependencies satisfied or not,
- A **list** used for keeping track of cells that have their dependencies satisfied but are yet to be calculated - a *pending cell list*, and
- A **list** acting as a cache. The presence of cells needed for a calculation in this list determines whether there is a cache hit or miss. Cells can be added in the head or tail of the list. After each step, the size of the list is checked. If it exceeds the cache size passed as an argument, the exceeding cells are removed from the tail of the list.

At first, the cell $(1, 1)$ (the only cell that has its dependencies satisfied) is inserted into the *pending list*. Afterwards, the simulator enters a loop that breaks when the pending list is empty, i.e. there are no more cells to be calculated and the algorithm has finished. Inside that loop, the simulator performs the parsing of the matrix. It begins by removing as many cells from the pending list as possible, to a maximum of a number equal to the number of workers. The position of the cells removed from the pending list is determined by the parsing method selected: In the case of diagonal, the cells are removed from the head of the list, whereas in the case of vertical, they are removed from its tail. In this way, the matrix is parsed in a manner adhering to the desired parsing method.

Following the removal of each cell from the pending list, the simulator checks whether its dependencies are in the cache list and updates the hit and miss statistics accordingly. If a dependency is not in the cache list, it is added in accordance to its position in relation to the cell that is being calculated: If (m, n) is the position of the cell and $(m-1, n-1)$ the position

of the dependency, the dependency is added to the tail of the cache list. Otherwise, it is added in the head. In this way, diagonal dependencies that are used for only one calculation are more likely to be removed from the cache. Subsequently, the simulator changes the cell status on the matrix to 'calculated', checks if the cells $(m+1, n)$ and $(m, n+1)$ have their dependencies satisfied (if they exist) and adds them in the pending list. This procedure acts as a step in the algorithm.

After each step, the simulator prints current and cumulative hit and miss statistics on a log file for reference. Each log file also contains data on which cells were brought to cache and which cells were removed from the cache, as well as the status of the dynamic programming matrix with regard to the calculation status of the cells.

For our analysis, we created a bash script to run the simulator on a given set of data, collect the hit and miss results from its log files and create graphs with them using **gnuplot**. The data set of the simulations was as follows:

- A set of varying sequence input lengths, resulting in matrices with dimensions 6x3, 15x8, 45x24, 150x8 and 450x240.
- For each matrix size we considered a set of 5, 10, 60, 100 and 234 workers.
- For each simulation regarding the number of workers, the cache size was determined as $[S * num_{workers}]$, where $S \in (0.8, 1.5, 2, 4, 8)$. This way, we can inspect how the cache performs in relation to the number of working units, while excluding cache values that would be too small or too large and thus unnecessary for our study.
- Each simulation was performed under both diagonal and vertical parsing methods.

This data set will help us to understand the behavior of both parsing methods in varying cache sizes and number of workers that varies from equal to much smaller order in relation to the number of matrix cells. For each simulation on a mxn sized matrix, we expect $3*(m-1)*(n-1)$ cache accesses. That is because the cells on the first row and column of the matrix do not need to be calculated and each cell calculation depends on data from 3 neighboring cells.

A total of 250 simulations were run on the cache simulator, as a result from the input dataset provided. We collected the cumulative cache hit and miss results and grouped them by the number of workers and the size of the matrix. Then we produced graphs showing the hits and misses of both parsing methods under the same figure for each group, for us to visualize

and better understand the performance of each method. This procedure resulted in 25 graphs. In this section we present the most important ones. We have also included the exact hit and miss statistics for all graphs in section 7.2 of the Appendix, together with the number of cells of the matrix and the cache accesses.

We expect the misses in a simulation on a mxn matrix to be at least $m + n - 1$. That is, because the cells in the first row and column of the matrix are not present in the cache when the parsing begins, therefore requesting them will result in a cache miss at least once. In the case where the number of workers is relative to or bigger than the matrix dimensions, both parsing methods have similar or equal performance. This happens because when the number of workers is comparable to the dimensions of the matrix, both methods follow a similar parsing pattern, with negligible deviations from each other. Moreover, for a cache big enough, the methods achieve the minimum number of misses, $m + n - 1$.

However, when the number of workers is smaller than the matrix dimensions, the parsing methods perform quite differently. In figure 4.7a of a simulation with 5 workers and a 15x8 matrix, we can see that when the cache size increases, the diagonal method quickly achieves marginally more cache hits than the vertical one. This is however not the case in simulations shown in figures 4.7e and 4.7f, where the matrix size gets bigger than the number of workers (and subsequently the cache size, as set in the simulation parameters). Here, the diagonal method strives to reach the performance of the vertical one and manages to achieve a marginally better hit/miss rate only with the maximum cache size.

Moreover, in figure 4.7b, when the cache size becomes larger than the number of workers, the diagonal method performs worse than the vertical one and in figure 4.7c it fails to have more hits than misses under any cache size tested. This is due to the fact that the diagonal method needs to keep in the cache, cells across the previous reverse diagonal of the matrix, as shown in figure 4.9 - in the case the matrix has a diagonal with size similar to the cache, there would be no issue. However, when working on larger matrix, it will result in high miss rate and overall poor performance. In contrast, the vertical method performs in a consistent manner, no matter the size of the matrix - thus, making it a better choice for our implementation.

4.4 Simulation Results Analysis

When parsing a mxn matrix with k workers with the vertical method, when traversing a set of columns, the calculations the workers $2, ..., k$[1] need to perform for each step depend solely on data calculated in the previous two steps. The only exception would be the first steps when parsing the set, where the workers will need to retrieve data from the first row of the matrix. Only worker 1 needs data calculated from the traversal of the previous set of columns, as shown in figure 4.8. Those dependencies were calculated by worker k while traversing the previous set of columns. This procedure results in a high rate of data reuse and is the cause of the improved efficiency of the method over the diagonal one.

The poor performance of the diagonal method is due to the fact that this method needs to keep in the cache, cells across the previous reverse diagonal of the matrix, as shown in figure 4.9. In the case the matrix has a diagonal with size similar to the cache, there would be no issue. However, when working on larger matrix, it will result in high miss rate and overall poor performance. In contrast, the vertical method performs in a consistent manner, no matter the size of the matrix - thus, making it a better choice for our implementation. Considering this observation, we proceed to make the following statement for the vertical parsing method:

> In a vertical parsing method, for the implementation to be efficient, the cache size needs to be *at least* $2*n_{workers}+2$. This accommodates for storing the dependencies from the two previous steps of calculations. Any additional space can act as a 2nd level cache to be used towards storing dependencies generated by the n-th worker that can be used by the first worker when parsing the next set of columns.

When a group of n workers traverse a single set of columns, their data dependencies equals to $3*n$, as each cell (j,k) depends on data from cells $(j-1,k)$, $(j-1,k-1)$ and $(j,k-1)$; those cell dependencies though overlap, which results in $2*n+1$ discrete dependencies, as shown in figure 4.10. However, when the group of workers splits into two sets of columns, as shown in figures 4.5e, 4.5f and 4.5g, we essentially have two groups of k and $n-k$, $k<n$ workers that have different sets of dependencies. In this case, the number of dependencies equals to $2*k+1+2(n-k)+1 = 2*n+2$.

The analysis of the Needleman-Wunsch algorithm we presented in this chapter helped us understand its constraints and challenges imposed in an attempt to design a parallel implemen-

[1]The number of each worker refers to the column the worker traverses under each set, i.e. worker 1 traverses the leftmost column of the set, worker 2 traverses the one on its right, etc.

tation of it on an FPGA platform. Based on the results we collected from this analysis, we can propose the following directions as a guide for our implementation:

- A set of n workers to perform the parsing of the matrix.
- The workers would perform the traversal in a way adhering to the *vertical parsing method*, thus achieving better cache efficiency.
- A cache of size $2 * n + 2$ to accommodate for data dependencies required by the workers.
- An additional *2nd level cache* outside of the IP to accommodate for data required by the first worker.
- A scheduler to assign the workload to workers and manage the cache and inputs/outputs.
- An interface implementation for the design to access the 2nd level cache for data retrieval and writing, as well as sending data of calculated cells to storage outside of the design.

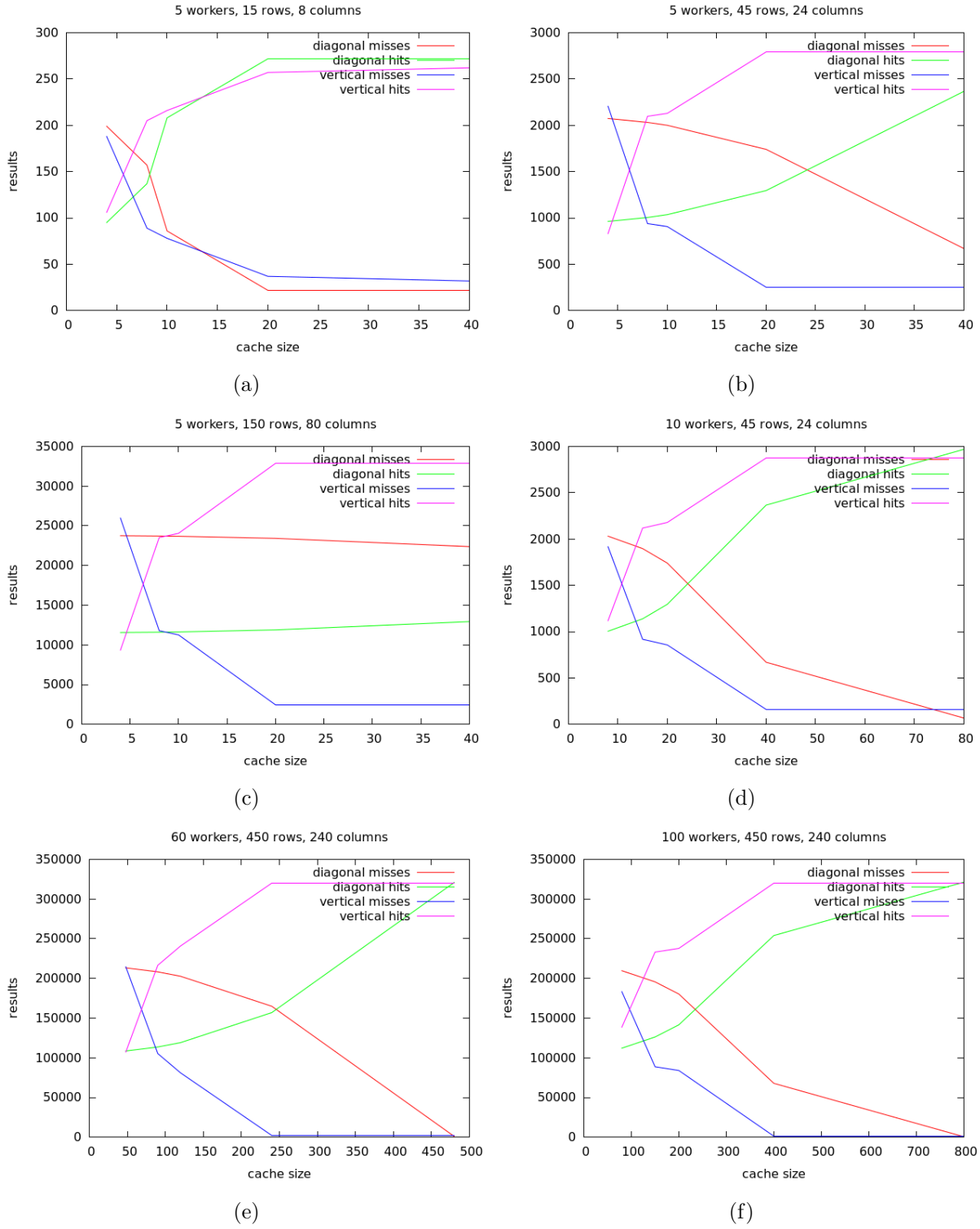


Figure 4.7: Cache simulator results

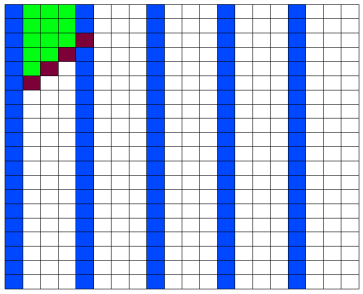


Figure 4.8: Data dependencies for the first worker in a vertical parsing method: When traversing this 20x20 matrix with 4 workers, the first worker will need data from the cells colored in blue.

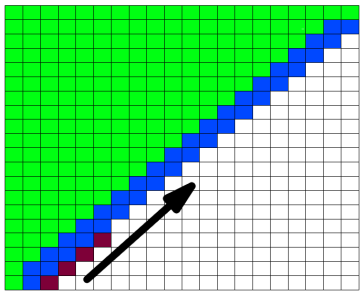


Figure 4.9: Data dependencies in a diagonal parsing method: On this 20x20 matrix, the workers assigned on the red cells will continue as indicated by the arrow. The blue cells in the previous diagonal need to be present in cache, for the dependencies of the next steps to be satisfied.

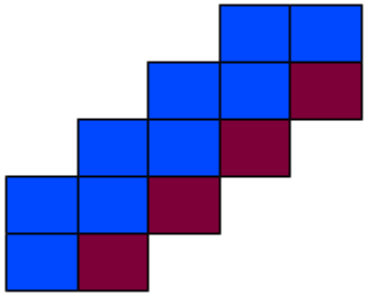


Figure 4.10: Data dependencies for a set of workers: In this example, 4 workers have $2*4+1 = 9$ discrete dependencies.

Chapter 5

Algorithm Implementation on Vivado HLS

In this chapter, we present in detail the steps we followed towards implementing the Needleman-Wunsch algorithm on Vivado HLS. We begin with a brief description of the tool and then proceed to present the source code we will be using. Subsequently, we demonstrate two implementations of the algorithm, together with performance results with different sets of directives.

5.1 Retrieving a Code Source

The first step towards implementing the Needleman-Wunsch algorithm on the Vivado HLS was to retrieve the source code of an existing implementation. We required the code to be written in a high-level programming language by a reputable source. In this way, we could ensure the correctness of the code and proceed to import it to Vivado HLS for our study.

The first source we considered was the algorithm implementation by the National Center for Biotechnology Information (NCBI), as a part of the NCBI C++ Toolkit software suite [15]. This suite also includes the well-known BLAST (Basic Local Alignment Tool) algorithm [16] and provides a framework of libraries aimed at supporting bioinformatics services used inside NCBI.

Despite our efforts to extract the Needleman-Wunsch implementation from the NCBI Toolkit, its tight integration with the suite and the multitude of additional code modules it depended on would compel us to perform quite a few alterations to the original code. This procedure that would defy our initial purpose to preserve the integrity of the source code we would import to

Vivado HLS. Therefore we ultimately deemed the NW implementation from the NCBI Toolkit unfeasible for use to our study.

The next source we considered was an implementation of the algorithm by Dr. Y. Zhang at the University of Michigan. [17] The source is provided in both Fortran and Java and follows a simple and straightforward approach to implementing the NW algorithm. We selected the Java version and converted it into C++ to achieve compatibility with Vivado HLS, making as few alterations as possible, in order to keep the error margin at a minimum. The code also included an alternative implementation of the algorithm that we did not include in our conversion. After converting the code, we compared the output of both programs to verify its correctness.

5.1.1 Presentation of the Algorithm

The algorithm accepts two sequences of amino acids in FASTA [1] format, and negative integers for gap introduction and extension penalties. It then fills the dynamic programming matrix using a BLOSUM62 matrix (listed in section 7.3 of the Appendix) when calculating the score. When the dynamic programming part is completed, it traces back the matrix and prints the global alignment of the sequences, together with the alignment score. Next we present in detail all the matrices, arrays and variables the algorithm uses:

1. The pair of sequences are read into the strings `f1` and `f2`. The gap open penalty is read into `gap_open` and the gap extension penalty into `gap_extn`.

2. The `imut` matrix is initialized as a 23x23 BLOSUM62 matrix, together with the string `seqW` holding the amino acid sequence `ARNDCQEGHILKMFPSTWYVBZX` to match the matrix.

3. Each of `f1`, `f2` is transformed into integer arrays `seq1` and `seq2`, respectively. Each cell of `seq1`, `seq2` holds the position of `seqW` in which the corresponding amino acid in `f1`, `f2` appears.

4. A two-dimensional `score` matrix is initialized, so that `score[i][j] = imut[seq1[i]][seq2[j]]`, i.e. each cell `score[i][j]` holds the BLOSUM62 score of matching the amino acids in `f1[i]` and `f2[j]`.

[1] A FASTA-formatted text file consists of one or more amino acid sequences, with a preceding line for each sequence that starts with ">" and is followed by a unique sequence identifier and an optional description. For our implementation, we only expect one sequence per FASTA file and ignore the line beginning with ">".

5. The matrices `val`, `idir`, `preV`, `preH`, `jpV` and `jpH` are initialized, each with dimensions `f1.length` x `f2.length`.

   - `val` holds the dynamic programming results. Per the algorithm's specification, it is initialized as `val[0][0] = gap_open`, `val[0][j] = val[0][j - 1] + gap_extn` for $j > 0$, `val[i][0] = val[i - 1][0] + gap_extn` for $i > 0$ and `val[i][j] = 0` for $i, j > 0$.
   - `idir` acts as a *traceback* matrix. Because it is an integer matrix, diagonal is 1, horizontal is 2 and vertical is 3.
   - The `prevV` and `prevH` matrices hold score data on already opened vertical and horizontal gaps, respectively.
   - The `jpV` and `jpH` act as support matrices to record the size of each extended vertical and horizontal gap, respectively.

6. The algorithm performs the dynamic programming part to evaluate the alignment. This is implemented as a set of two nested `for` loops for traversing each cell of `val` sequentially. The formula the algorithm follows is:

$$\texttt{val[i][j]} = \max \begin{cases} \texttt{val[i - 1][j - 1] + score[i][j]} \\ \texttt{val[i - 1][j] + gap\_open} \\ \texttt{preH[i - 1][j] + gap\_extn} \\ \texttt{val[i][j - 1] + gap\_open} \\ \texttt{preV[i][j - 1] + gap\_extn} \end{cases}$$

   This considers diagonal, vertical and horizontal scores, together with gap opening or extension penalties. Together with `val`, the matrices `idir`, `prevV`, `prevH`, `jpV` and `jpH` are also updated to store traceback data and gap extension sizes and penalties.

7. After the dynamic programming matrix is filled, the algorithm proceeds to perform the traceback operation and store the aligned sequences and final score of the alignment to a text file.

5.2 Steps on Implementing the Algorithm

The implementation of the Needleman-Wunsch algorithm we chose for our study follows a simple and straightforward sequential approach, with no optimizations towards parallelism or platform-specific code, making it suitable for our case. In this section, we present our steps

towards implementing the algorithm on Vivado HLS, optimizing it through the use of directives and then attempting a different implementation for which we applied the findings we collected from the previous chapter.

5.2.1 Implementation Using the Provided Code

For our first implementation attempt, we transfered the provided source code to Vivado HLS and focused on using the directives the suite provides to optimize its performance, while keeping any changes in the code to a minimum. In this way, we were able to observe how Vivado HLS performs on a non-optimized sequential code, what level of parallelism can be achieved and what the margins of speed-up are. Initially, we created the header file `datatypes.h`, where we set the struct `bus_out`, which returns the aligned sequences and score the algorithm calculates. Essentially it acts as an output interface. The data structure `bus_out` includes:

- `fina_score` to carry the alignment score,
- `L_ali`, `L_id` and `identity` to carry sequence identity information,
- the arrays `sequenceA`, `sequenceB` and `sequenceM` that hold the aligned sequences (`sequenceM` is used to provide spacing and helps identify the matches between `sequenceA` and `sequenceB`).

Inside `datatypes.h` we defined the constant `MAXSIZE` that we use to set the maximum size of the arrays of the algorithm. As Vivado HLS does not allow defining arrays with size not known at compile time, we need to have a maximum, predefined size for them. Additionally, we used the `ap_int<val>` and `ap_uint<val>` type primitives provided by Vivado HLS to define (un)signed integers with a custom bit length of `val`, in order to minimize the size of arrays that are used in the algorithm. We removed the write-to-file part of the code. Instead, for returning data, we set the function to return a struct of type `bus_out`, which will later be implemented as part of the interface. We also changed the arguments the function accepts (which will later be implemented as the interface of the core) to the following:

- `char f1[MAXSIZE]` that contains the first input sequence,
- `ap_uint<val> f1_length` that contains the length of the first input sequence[1],
- respectively, `char f2[MAXSIZE]` and `ap_uint<val>` for the second input sequence,

[1] For a successful implementation, we should use an appropriate value for `val` considering the value of `MAXSIZE`.

- `ap_int<5> gap_open` and `ap_int<5> gap_extn` to define the penalties for gap opening and gap extension.

In order to observe the impact of the directives on the design, we ran two of tests: One with input sequences of lengths 10 and 50, and one of 300 and 500. Therefore, for every loop with bounds depending on the size of the sequences, we used the directive `LOOP_TRIPCOUNT`[1] to inform Vivado HLS of the number of iterations expected for each loop and updated the `MAXSIZE` constant to 100 and 1000, respectively. Additionally, we set `val` to 7 and 10, respectively.

After the aforementioned steps, we decided to split the workflow in two parts: For the first, we synthesized the code without the use of any optimization directives, in order to have an unoptimized set of results for reference and for the second one, we gradually applied a list of directives and monitored their impact on the design. This procedure was carried out using the **solutions** perspective offered by Vivado HLS; each solution refers to a designer-defined list of directives that can be implemented for synthesis, making implementations with different directive sets easy to manage and perform comparisons on their results.

We ran each implementation on a system running Ubuntu Linux 16.04, with 4GB of RAM and a 2.13GHz Core 2 Duo processor. The Vivado HLS version we used was 2016.2. We defined xcku035-sfva784-3-e as a target device which corresponds to a Xilinx Kintex UltraScale FPGA. This device package was the most resourceful available in the Vivado HLS device selection list, with 25391 SLICEs, 203128 LUTs, 406256 FFs, 1700 DSPs and 1080 BRAMs. We set the clock period to be 10ns.

The synthesis of the unoptimized code took 33.15 seconds to complete. Vivado HLS automatically bound the ports of the core to the following interfaces:

- `fina_score`, `L_ali`, `L_id` and `identity` to `ap_vld`,
- `sequenceA`, `sequenceB`, `sequenceM` `f1` and `f2` to `ap_memory`,
- `f1_length`, `f2_length`, `gap_open` and `gap_extn` to `ap_none`.

Additionally, it correctly identified the arrays `imut` and `seqW` as read-only and implemented them as distributed ROMs, whereas it implemented `val`, `idir`, `preV` and `preH` as block RAMs and `seq1` and `seq2` as distributed RAMs.

[1] As stated in the list of directives, `LOOP_TRIPCOUNT` is merely used for design analysis and does not impact synthesis.

After synthesizing an unoptimized implementation of the algorithm, we proceeded to apply a list of directives to the code: At first, we focused on optimizing the loops inside the algorithm: For each loop, we applied the `PIPELINE` directive. Vivado HLS automatically changed the Initiation Interval from 1 to 2 to satisfy carried dependency constraints for all occurrences. Subsequently, in an attempt to improve the efficiency of the resource utilization, we applied directives on the interface: We set `f1_length`, `f2_length`, `gap_open` and `gap_extn` to `ap_stable`, because we expect those variables to have a constant value throughout the execution of the algorithm. Additionally we attempted to set `sequenceA`, `sequenceB` and `sequenceM` to `STREAM`; however Vivado HLS informed that those arrays are accessed in a non-sequential manner, therefore it could not implement them as FIFOs used for `STREAM`. As a result we set them to `ap_memory`. As a next step towards improving latency, we applied the `UNROLL` directive for each loop. Vivado HLS failed to implement it on the double-loop of the dynamic programming part of the code and the traceback loops because of their variable bounds; however it successfully implemented it on the initialization loops of `seq1` and `seq2`, which were iterating on constant bounds. Furthermore, we attempted optimizing the array accesses by partitioning them, using the `ARRAY_PARTITION` directive. However, using the `cyclic` and `block` option showed no improvement on the results and `complete` caused Vivado HLS to exhaust the available memory of the system and stop responding. Therefore we ultimately removed the `ARRAY_PARTITION` directives altogether. Additionally, we attempted to use the `DATAFLOW` directive on the design. Vivado HLS however informed us that did not discover any possible data sharing optimizations between the loops of the algorithm and thus failed to implement the directive.

The latency results of our implementations are shown in tables 5.1 and 5.2, for both tests. We can observe that the `PIPELINE` directive caused a vast improvement in latency, especially on the test with the larger set of inputs. Additionally, the use of the `UNROLL` directive further improved latency, although to a lesser extent, while the `ap_stable` interface directive showed no improvement regarding latency. However, it allowed Vivado HLS to perform optimizations regarding the use of available resources, which caused an improvement in resource utilization, as shown in tables 5.3 and 5.4. In contrast, both `PIPELINE` and `UNROLL` directives caused a slight increase in resource utilization. In figure 5.1 we can observe the latency speedup the use of directives provided to the design with respect to the latency observed without the use of directives. In both matrix sizes, we observe a speedup factor of about 2x when using both `PIPELINE` and `UNROLL` directives.

Directives	Latency (clock cycles)	
	Synthesis	RTL Simulation
None	6526 (2650)	7007
+PIPELINE	4817 (1002)	5314
+ap_stable	4817 (1002)	5314
+UNROLL	3317 (1002)	3814

Table 5.1: Latency results on a 10x50 matrix

Directives	Latency (clock cycles)	
	Synthesis	RTL Simulation
None	946606 (751500)	952035
+PIPELINE	494307 (300002)	499354
+ap_stable	494307 (300002)	499354
+UNROLL	474707 (300002)	479350

Table 5.2: Latency results on a 300x500 matrix

Directives	Resources			
	BRAM_18K	DSP48E	FF	LUT
None	24	8	3048	5425
+PIPELINE	24	9	3098	5530
+ap_stable	24	9	2938	5530
+UNROLL	24	9	3011	5854

Table 5.3: Resource utilization on a 10x50 matrix

Directives	Resources			
	BRAM_18K	DSP48E	FF	LUT
None	1963	8	3167	5636
+PIPELINE	1963	9	3241	5782
+ap_stable	1963	9	3051	5782
+UNROLL	1963	9	3116	6100

Table 5.4: Resource utilization on a 300x500 matrix

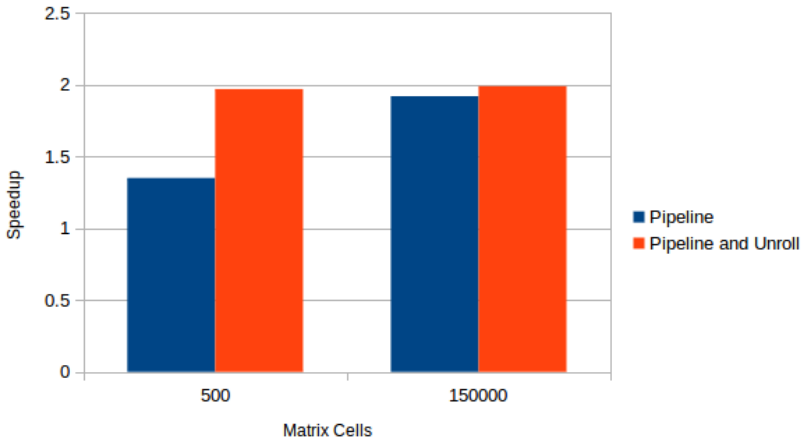


Figure 5.1: Implementation speedup with different sets of directives

We also observed, however, that although the use of directives improved the overall latency of the design significantly, the latency of the dynamic programming part (shown in parentheses in the Synthesis column) would not drop below a certain value. This indicates that the dynamic programming matrix is still accessed sequentially (each cell calculation requires two clock cycles and for a matrix of size m xn the latency of its traversal would be $m*n*2+2$, together with an initiation interval of two cycles). Therefore we concluded that the sole use of directives would not be sufficient to parallelize the Needleman-Wunsch algorithm in Vivado HLS. Additionally, for the larger test, the usage of BRAMs (memory elements) grew to 1963, making the design too large for a pure FPGA implementation (the device we used only had 1080 BRAMs). Therefore, we concluded that the sole use of directives would not be sufficient to parallelize the Needleman-Wunsch algorithm and implementing the whole design on an FPGA would be unfeasible for large inputs.

5.2.2 Implementation Based on the Analysis Results

In this section, we present our attempt to implement the Needleman-Wunsch algorithm on Vivado HLS considering the findings of the theoretical analysis. Therefore we assumed the following:

- Due to their variable (and likely large) size, the matrices `val`, `idir`, `preV`, `preH`, `jpV` and `jpH` would be stored outside of our implementation and accessed through an interface.

- We solely focus on the dynamic programming part of the algorithm, thus expecting array initializations and traceback to be performed outside of our implementation.
- We would implement a 1st level cache inside our design.
- Due to its arbitrary size, we would expect a 2nd level cache to be implemented outside of our design and mediate between our implementation's interface and the dynamic programming matrices. For our study, we did not proceed to implement it.
- A scheduler implemented in the top level function of our design would assign workload to the workers in order for them to parse the matrix with the vertical parsing method.

The block diagram of our suggested implementation is shown in figure 5.2. The workers, 1st level cache and scheduler reside inside the Vivado HLS core. Our design interfaces with a memory controller that accesses a 2nd level cache and the dynamic programming matrices. The memory controller, as well as the 2nd level cache could be implemented inside an FPGA development board and take advantage of any available memory module (e.g. RAM) to increase data locality. Additionally, the memory controller would interface with an external storage (e.g. CPU-controlled hard drive) to store the dynamic programming array data. The external storage would also contain the input sequences `seqA` and `seqB`. We now proceed to provide details regarding the cache implementation and the interface.

Cache Implementation As stated in the previous chapter, we considered the implementation of a first level fully associative cache with size $2 * n_{workers} + 2$. Each cache cell holds the cell value, the row and column of the cell acting as a tag, and a valid bit variable. As shown in figure 5.3, a cache cell distributes data to two workers vertically and horizontally (with the exception of the bottom-left and top-right cell) and to one worker diagonally. Moreover, when a cache cell is accessed by a worker diagonally, its value will not be read again, due to the parsing pattern. Therefore its value can be flushed to the external storage and the cell becomes empty. This is also the case for the cell carrying the vertical dependency of the n-th worker. However, when its data is flushed, it is expected to stay in the second-level cache, for faster access when processing the next set of columns.

We implemented the cache as an array with $2 * n_{workers} + 2$ cells. We then used the `ARRAY_PARTITION` directive in order to decompose it into individual elements and unrolled the loops that traverse it with the `UNROLL` directive in an attempt to fully parallelize the array

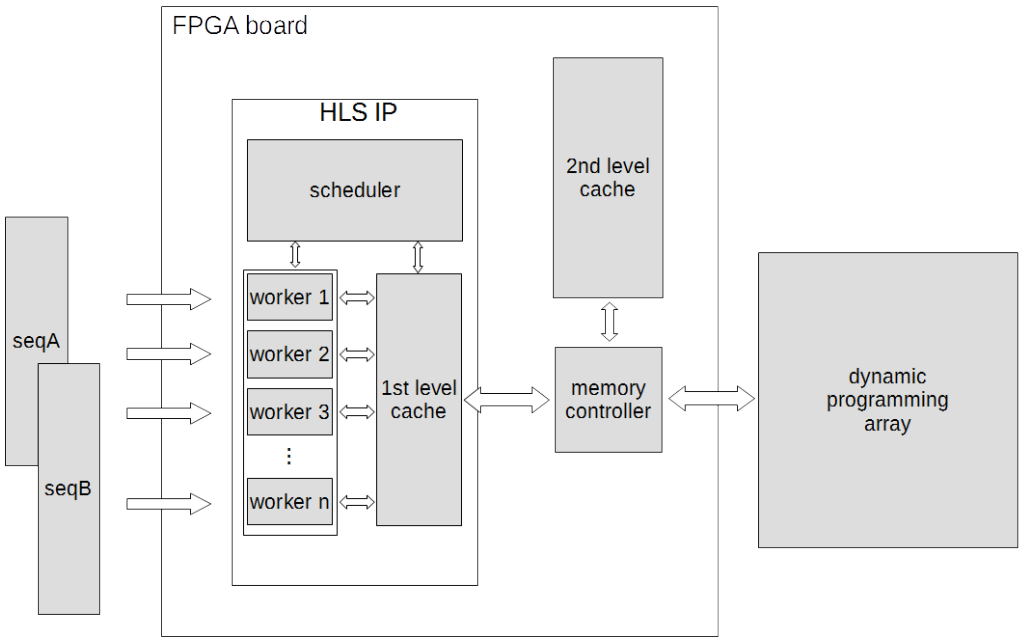


Figure 5.2: Block diagram of the parallel Needleman-Wunsch implementation

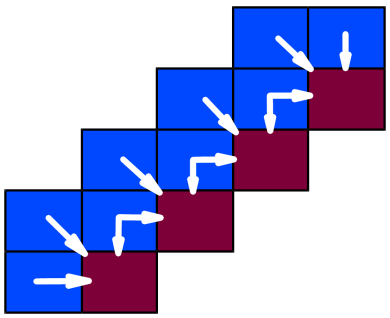


Figure 5.3: Cache data distribution among 4 workers: The cells that are being evaluated by the workers are indicated with red and the cells carrying dependencies are indicated with blue.

accesses. The implementation we aimed at is shown in figure 5.4. Each cache entry holds data the size of which can be defined by the designer before synthesis, in order to accommodate for different value ranges.

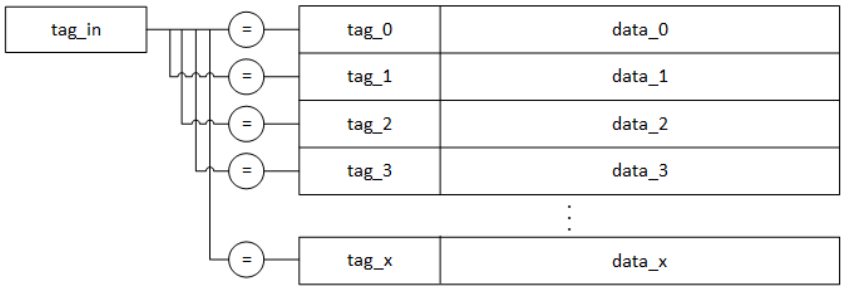


Figure 5.4: Block diagram of the fully associative 1st level cache

Workers For our implementation, we defined the `Worker` class. This class contains the part of the code inside the dynamic programming loop of the original implementation. The workers accept the values required to perform the computation as arguments and return the calculated `val`, `idir`, `preH`, `preV`, `jpH` and `jpV` values inside a struct.

Scheduler The scheduler assigns the workload to the array of workers, establishes the communication between the cache and the workers and implements the interface of the core. In our case, the scheduler uses two nested loops to parse the dynamic programming matrix: The outer runs for as long as there are utilized workers (due to the uncertainty of the size of the inputs); the inner is used to assign the workload to each worker - where we would concentrate our efforts for parallelization.

Interface In addition to the preexisting ports `f1`, `f2` (which we now implemented as pointers to accommodate for unknown input sequence lengths), `f1_length`, `f2_length`, `gap_open` and `gap_extn`, we introduced an `array_interface` of type `nw_data_t *`, a struct pointer implementing the communication of the core to the external storage. Initially we attempted to implement a pointer array with size $2 * n + 2$ for each cache cell to communicate independently. However, the Vivado HLS would not support pointer implementation inside an array; therefore we resulted to using only one instance of the `array_interface`. This would force the cache cells to share one bus when reading or flushing data, which would cause performance issues and limit throughput.

As a first step, we applied the `LOOP_UNROLL` directive on the inner loop of the scheduler, in order for the workers to parse the dynamic programming matrix in parallel. We additionally set the ports `f1`, `f2` and `array_interface` to use the `ap_bus` interface, as it allowed non-sequential accesses through pointers.

For our test, we set the number of workers to 8 and used a sample set of sequences both with length of 50. We ran both Synthesis and RTL Simulation: The simulation process completed after a day of running and the results are shown in table 5.5. We would attribute those values to the lack of interface ports as well as the complexity of the design, as we essentially described a hardware implementation. Our attempts to apply any pipeline directives, as well as increasing the number of workers failed due to Vivado HLS exhausting the available memory of the system and halting. Table 5.6 presents the estimated resource utilization of the design.

Directives	Latency (clock cycles)	
	Synthesis	RTL Simulation
`UNROLL`	563157	442946

Table 5.5: Latency results of the implementation based on the analysis of the algorithm, on a 50x50 matrix

Directives	Resources			
	`BRAM_18K`	`DSP48E`	`FF`	`LUT`
`UNROLL`	2	49	17124	39648

Table 5.6: Resource utilization of the implementation based on the analysis of the algorithm, on a 50x50 matrix

Chapter 6

Conclusions - Future Work

In this thesis, we presented the Needleman-Wunsch algorithm as a prime example of an application implementing the dynamic programming methodology and proceeded to perform an in-depth analysis of it under the scope of parallelism. Additionally, we developed and presented a cache simulator in Java, which helped us observe and evaluate the performance of the matrix parsing methods in regard to cache hits and misses. Our findings helped us form a solid set of suggestions for its parallelization, considering both spatial and temporal constraints and limitations posed by the distinct characteristics of the algorithm. We subsequently implemented two On our subsequent attempt to accelerate the algorithm on Vivado HLS, we made the following observations:

- Implementing an algorithm using Vivado HLS is a straighforward procedure that provides a higher level of abstraction from hardware description languages, which speeds up the design process.
- The use of optimization directives was proven to be beneficial for the performance of the design. Especially, the `PIPELINE` directive on loops with variable bounds the `UNROLL` on loops with static bounds and no carried dependencies can improve the latency compared to not using them by almost a factor of 2, per our observation.
- However, the use of directives can also cause long synthesis and RTL simulation times and in some cases it might not provide the desired performance improvement.
- Additionally, Vivado HLS might fail to parallelize a design, despite the use of directives. This is however to be expected to a certain degree, as successful parallelization requires a good understanding of the mechanics and limitations of the algorithm.

- In larger and more complicated designs, Vivado HLS suffers from long implementation times and exhausts the available system memory, which might cause unexpected termination of the program.

As future work of our study, we would suggest proceeding to download the design on an FPGA board and evaluate its performance on actual hardware. This would help us validate the correctness of the algorithm and collect more accurate results regarding its performance. Additionally, we would compare our implementation against designs developed purely in Hardware Description Language, as well as software-based implementations for execution in a CPU, and compare their performance results. In this way, we would be able to observe the efficiency and performance of the design synthesized by the Vivado HLS compared with software and hardware-based solutions. Moreover, we would also compare the results of our design to those of other dynamic programming algorithm designs, especially the Smith-Waterman algorithm that is used for local sequence alignment and shares the same algorithmic principles with Needleman-Wunsch. This would provide useful information regarding any performance deviations occurring due to differentiations between the algorithms.

Finally, we would propose an alternative approach for the parallel Needleman-Wunsch implementation: Although attempting to implement the whole design on Vivado HLS yielded unsatisfactory results, we would consider using the tool to create RTL designs for the worker modules and then proceed to import them in a Vivado VHDL design. With this method, a scheduler and a cache can be designed with more precision in VHDL and communicate with the workers through an appropriate interface. This would enable us to compare the results of such solution with those presented in this study, ultimately coming to conclusions on an optimal design methodology.

References

[1] Power4: The First Multi-Core, 1GHz Processor https://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/ 1

[2] Colella P.: Defining Software Requirements for Scientific Computing 3

[3] Asanovic K., Bodik R., Catanzaro B., Gebis J., Husbands H., Keutzer K., Patterson D., Plishker W., Shalf J., Williams S., Yelick K.: The Landscape of Parallel Computing Research: A View from Berkeley 1, 3

[4] The OpenDwarfs Benchmark Suite https://github.com/vtsynergy/OpenDwarfs 6

[5] Krommydas K., Feng W., Antonopoulos C., Bellas N.: OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures 6

[6] Verma A., Helal A., Krommydas K., Feng W.: Accelerating Workloads on FPGAs via OpenCL: A Case Study with OpenDwarfs 6

[7] Georgopoulos K., Chrysos G., Malakonakis P., Nikitakis A., Tampouratzis N., Dollas A., Pnevmatikatos D., Papaeftathiou Y.: An Evaluation of Vivado HLS for Efficient System Design 6

[8] Karras K., Blott M., Vissers K.: High-Level Synthesis Case Study: Implementation of a Memcached Server 6

[9] Diamantopoulos D.: Cross-Layer Rapid Prototyping and Synthesis of Application-Specific and Reconfigurable Many-accelerator Platforms 6

[10] Cornu A., Derrien S., Lavenier D.: HLS Tools for FPGA: Faster Development with Better Performance 7

[11] Needleman S., Wunsch C.: A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins 9

[12] Musso G.: CSC2427 - Course Notes: Alignment (Needleman-Wunsch, Smith-Waterman) http://www.cs.toronto.edu/~brudno/csc2427/Lec7Notes.pdf 10, 12

[13] Frigo M., Strumpen V.: Cache Oblivious Stencil Computations 13

[14] Xilinx Inc.: Vivado Design Suite User Guide: High-Level Synthesis (UG902) 5

[15] NCBI: A Needleman-Wunsch implementation for the BLAST suite https://www.ncbi.nlm.nih.gov/IEB/ToolBox/CPP_DOC/lxr/source/src/algo/align/nw/nw_aligner.cpp 27

[16] Altschul S., Gish W., Miller W., Myers E., Lipman D.: Basic Local Alignment Search Tool 27

[17] Zhang Y.: NW-align: A Needleman-Wunsch implementation in Fortran and Java http://zhanglab.ccmb.med.umich.edu/NW-align 28

[18] Henikoff S., Henikoff J.: Amino acid substitution matrices from protein blocks

Chapter 7

Appendix

7.1 A: A List of HLS Directives

The following is a cumulative list of directives as used in the Vivado High Level Synthesis software, together with an explanation of their usage.

1. **ALLOCATION**: Specifies a limit for the number of operations, cores or functions used. This can minimize the sources used for the RTL design, but might force the sharing of hardware resources and increase overall latency.

   Options:

   - **limit**: Refers to the maximum number of allowed instances of the block
   - **type**:
     (a) **core**: The directive refers to a readily available Xilinx core (the designer can pick it from a list).
     (b) **function**: The directive refers to a function defined in the C source code (the designer must specify its name)
     (c) **operation**: The directive refers to an operator, such as add, sub, etc. (the designer can pick it from a list).

2. **ARRAY_MAP**: Combines multiple smaller arrays into a single larger array, which can then be targeted to a single memory resource.

Options:

- **instance**: Specifies the name of the larger array to be used. If multiple directives refer to the same larger array, it gets populated with the same order as they appear.
- **mode**:
  (a) **horizontal** (default): The smaller arrays get concatenated to form a larger array with more elements and words with the same size as them.
  (b) **vertical**: The larger array has the same number of elements as the smaller ones, but longer words.
- **offset**: Specifies an integer value indicating the absolute offset in the target instance for current mapping operation. This applies to horizontal mode only and is optional, as Vivado HLS can automatically give it a value that will not cause overlaps.

3. **ARRAY_PARTITION**: Splits an array into smaller arrays or individual elements. It can improve the throughput of the design, at the cost of extra resources.

Options:

- **type**:
  (a) **block**: Smaller arrays holding consecutive blocks of the original array are created.
  (b) **cyclic**: Partitioning creates smaller arrays by interleaving elements from the original array.
  (c) **complete** (default): The array is decomposed into individual elements.
- **factor**: Specifies the number of arrays the original array will be partitioned into.
- **dimension**: In multi-dimensional arrays, it specifies the dimension the directive refers to. 0 is used to have the directive applied to all dimensions.

4. **ARRAY_RESHAPE**: Combines array partitioning with vertical array mapping to create a single new array with fewer elements but wider words

Options: As this directive is essentially a combination of **ARRAY_PARTITION** and vertical **ARRAY_MAP**, its options are the same as those of **ARRAY_PARTITION**.

5. **CLOCK**: Used to apply a clock to the specified function. Only SystemC supports multiple clocks.

6. **DATAFLOW**: This is a directive that orders Vivado HLS to analyze the dataflow between sequential functions or loops and attempt to create channels that allow consumer functions and loops to start operation before the producer functions or loops have completed. This allows a higher level of parallelization, which can improve throughput and decrease latency.

7. **DATAPACK**: Used to pack the fields of a struct into a single scalar. The first field of the struct takes the least significant sector of the scalar, etc.

   Options:

   - **instance**: Specifies the name of the scalar to be used.
   - **byte_pad**:
     (a) **field_level**: Pack each field of the struct on 8-bits boundary first, then pack the struct.
     (b) **struct_level**: Pack the struct first, then pack it on 8-bits boundary.

8. **DEPENDENCE**:Vivado HLS can detect variable dependencies within a loop and between different iterations of a loop. This directive is used to manually declare the presence (or not) of a dependence, in case Vivado HLS fails to detect it (or have a false-dependence).

   Options:

   - **variable/class(**array/pointer**)**: Specifies whether the directive refers to a particular variable (option **variable**), or a whole class of variables (option **class**). Those options are mutually exclusive.
   - **type**:
     (a) **inter** (default): The directive refers to variable operations within different loop iterations.

(b) **intra**: The directive refers to variable operations within the same loop iteration.

- **direction**: Refers to the nature of the dependence (RAW, WAR, WAW). This is relevant for loop-carry dependencies only.
- **distance**: Specifies the inter-iteration distance for array access.
- **dependent**: Sets whether the dependence is true or false (default).

9. **EXPRESSION_BALANCE**: Turns expression balancing on or off in a particular section. By default, Vivado HLS balances expressions automatically.

10. **FUNCTION_INSTANTIATE**: By default, all instances of a function, at the same level of hierarchy, use the same RTL implementation. This directive forces the creation of unique RTL implementation for each instance of the function, allowing them to be optimized. The designer must declare which function argument is to be specified as a constant.

11. **INLINE**: Removes a function as a separate entity and allows operations within the function to be shared and optimized more effectively with surrounding operations. An inlined function cannot be shared.

    Options:

    - **region**: All functions within the specified region are to be inlined.
    - **recursive**: Apply the directive recursively down the hierarchy. By default, the directive is applied on only one level.
    - **off**: Excludes the referenced region from being considered for automatic inlining.

12. **INTERFACE**: Specifies how RTL ports are created from the function description during interface synthesis.

    Options:

    - **mode**:
      (a) **ap_ack**: Implements an **acknowledge** port associated with the data port to acknowledge that the data was read or written.
      (b) **ap_bus**: Implements pointer and pass-by-reference ports as a bus interface.

(c) **ap_fifo**: Implements the port with a standard FIFO interface with active-on-low **empty** and **full** ports.

(d) **ap_hs**: Implements the data port together with **valid** and **acknowledge** ports, to provide a two-way handshake feature.

(e) **ap_memory**: Implements array arguments as a standard RAM interface that will appear as discrete ports in the Vivado IP integrator.

(f) **ap_none**: Does not implement any protocol.

(g) **ap_ovld**: Implements an output data port with an associated valid port.

(h) **ap_stable**: Does not implement any protocol and assumes that data is always stable after reset, to allow internal optimizations.

(i) **ap_vld**: Implements a data port with an associated **valid** port to indicate when the data is valid for reading/writing.

(j) **axis**: Implements all ports as an AXI4-Stream interface.

(k) **bram**: Implements array arguments as a standard BRAM interface that will appear as a single port in the Vivado IP integrator.

(l) **m_axi**: Implements all ports as an AXI4 interface. **config_interface** command can be used to specify 32-bit or 64-bit address ports and control any address offset.

(m) **s_axilite**: Implements all ports as an AXI4-Lite interface. Vivado HLS produces an associated set of C driver files during the Export RTL process.

- **register**: Registers the port and its related protocol signals and instructs the signals to persist until at least the last cycle of the function execution. Applies to the **ap_none**, **ap_ack**, **ap_vld**, **ap_ovld**, **ap_hs**, **ap_fifo** and **axis** modes.
- **depth**: Indicates the maximum size of the FIFO needed in the verification adapter that Vivado HLS creates for RTL co-simulation. It is required for pointer interfaces using **ap_fifo** and **ap_bus** modes.
- **offset**: Controls the address offset in AXI4-Lite and AXI4 interfaces. For AXI4-Lite, it specifies the address in the register map. For AXI4, it can be set as:

  (a) **off**: Do not generate an offset prot.

  (b) **direct**: Generate a scalar input offset port.

(c) **slave**: Generate an offset port and automatically map it to an AXI4-Lite slave interface.

- **bundle**: Groups function arguments into AXI ports. By default, Vivado HLS groups all function arguments specified as an AXI4(-Lite) interface into a single AXI4(-Lite) port. This option bundles together all arguments with the same **bundle** string and creates an RTL port with the same string as name.
- **clock name**: Specifies the clock name for the **s_axilite** mode.

13. **LATENCY**: Specifies the minimum or maximum latency on a region. Vivado HLS always aims for minimum latency and if it falls below the minimum set by the directive, it extends the latency to the specified value, potentially increasing sharing. Conversely, if it cannot match the maximum latency set, it increases effort to achieve the constraint and if it still fails, it issues a warning and produces a design with the smallest achievable latency.

14. **LOOP_FLATTEN**: Flattens nested loops into a single loop, saving clock cycles and potentially allowing for greater optimization.

    A perfect loop nest refers to a loop nest of which only the innermost loop has loop body content, there is no logic specified between the loop statements and all loop bounds are constant. A **semi-perfect loop nest** is similar to a perfect loop nest with the difference of having the outermost loop bound variable. An **imperfect loop nest** has inner loops with variable bounds, or loop body not exclusively inside the inner loop. This directive applies to perfect or semi-perfect loop nests and should be declared for the innermost loop in the loop hierarchy. If imperfect loop nest is the case, the designer might try to restructure the code or unroll the loops in the loop body to create a perfect loop nest, in order to apply the directive.

    Options:

    - **off**: Disables automatic loop flattening to a particular loop nest.

15. **LOOP_MERGE**: Merges all loops into a single loop, that reduces the number of clock cycles required in the RTL to transition between the loop body implementations and

allows the loops be implemented in parallel. For loops to be merged, if their bounds are variable, they must have the same number of iterations (if they are constant, the maximum value is used as the bound of the merged loop). Moreover, code between loops to be merged cannot have side effects and multiple execution of this code should generate the same results. Loops with both variable bound and constant bound or loops that contain FIFO reads cannot be merged.

Options:

- **force**: Forces loops to be merged, even when Vivado HLS issues a warning.

16. **LOOP_TRIPCOUNT**: In case the loop tripcount (the total number of iterations performed by a loop) cannot be automatically determined by Vivado in order to calculate its total latency, this directive can be used to declare the latter. This information is used for design analysis and does not have an impact on synthesis.

    Options:

    - **min**: Specifies the minimum latency.
    - **max**: Specifies the maximum latency.
    - **avg**: Specifies the average latency.

17. **OCCURENCE**: When pipelining functions or loops, this directive specifies that a particular code section is executed at a lesser rate than the code in the enclosing function or loop, allowing it to be pipelined at a slower rate and potentially shared within the top-level pipeline. Given that N is the number of times the enclosing function or loop is executed and M is the number of times the conditional region is executes, the ratio N/M must be an integer.

    Options:

    - **occuring cycle**: Specifies the N/M ratio.

18. **PIPELINE**: Specifies the details for function and loop pipelining.

Options:

- **II**: Specifies the Initiation Interval, the number of cycles needed for the pipelined function or loop to process new inputs. Default value is 1. Vivado HLS tries to meet this request, but the actual result might have a larger II.
- **enable flushing**: Implements a pipeline that can flush pipeline stages if the input of the pipeline stalls. This option implements additional control logic and thus demands greater area.
- **enable loop rewinding**: This applies only to loops and enables continuous loop pipelining, with no pause between one loop iteration ending and the next starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop is considered as initialization, executes only once in the pipeline and cannot contain any conditional operations.
- **disable loop pipelining**: Disables any automatic loop pipelining that might take place in the optimization process.

19. **PROTOCOL**: Specifies a region of the code (a protocol region) in which no clock operation is inserted by Vivado HLS, unless explicitly specified in the code. To declare a region of code, the designer must enclose it in braces and name it: **<name>:{}**

Options:

- **mode**:
  (a) **floating** (default): Floating mode allows the code corresponding to statements outside the protocol region to overlap with the statements in the protocol region in the final RTL. Although the protocol region remains cycle accurate, other operations can occur at the same time.
  (b) **fixed**: This mode ensures that there is no overlap.

20. **RESET**: Adds or removes reset signals for global or static variables.

Options:

- **off**: Whether it is ticked or not, reset signal is not/is generated, respectively.

21. **RESOURCE**: Specifies which resource (core) should be used to implement a variable in the RTL. The variable can be an array, an arithmetic operation or a function argument.

    Options:

    - **core**: Specifies the core to be used, as defined in the technology library.
    - **latency**: Specifies the latency of the core.
    - **port_map**: Specifies port mappings when using the IP generation flow to map ports on the design with ports on the adapter.

22. **STREAM**: Sets an array to be implemented as a FIFO for streaming data, which is a more efficient communication mechanism in a case the data in the array are consumed or produced in a sequential manner, in contrast to the default RAM array implementation. If a top-level function argument is specified as an **ap_fifo**, the array is identified as streaming.

    Options:

    - **off**: If the default channel is set to use a FIFO with the **config_dataflow** command, it globally implies a **STREAM** directive on all arrays in the design. This option allows streaming to be turned off on a specific array and default it back to using a RAM pingpong buffer based channel.
    - **depth**: Overrides the default FIFO depth specified by the **config_dataflow** command.
    - **dimension**: On multidimensional arrays, specifies which dimension the directive applies to. Default is 1.

23. **UNROLL**: This directive is used to unroll a loop, which minimizes the checks for exit conditions and can improve parallelism.

    Options:

    - **factor**: Specifies a non-zero integer as the unrolling factor. Leaving it blank suggests complete unrolling.

- **skip_exit_check**: On a loop with fixed bounds, no exit condition check is performed if the iteration count is a multiple of the factor. If the iteration count is not an integer multiple of the factor, Vivado HLS prevents unrolling and issues a warning that the exit check must be performed. On a loop with variable bounds, the designer must ensure that not performing an exit condition check will not cause problems.
- **region**: Unrolls all loops inside a particular region. In case the region itself is a loop, it is kept rolled.

7.2 B: Cache Simulator Hit and Miss Statistics

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
4	11	19	11	19
8	8	22	8	22
10	8	22	8	22
20	8	22	8	22
40	8	22	8	22

Table 7.1: 5 workers, 6 rows, 3 columns: 18 cells, 30 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
4	199	95	188	106
8	157	137	89	205
10	86	208	78	216
20	22	272	37	257
40	22	272	32	262

Table 7.2: 5 workers, 15 rows, 8 columns: 120 cells, 294 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
4	2073	963	2206	830
8	2031	1005	939	2097
10	2000	1036	907	2129
20	1740	1296	248	2788
40	670	2366	248	2788

Table 7.3: 5 workers, 45 rows, 24 columns: 1080 cells, 3036 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
4	23752	11561	25972	9341
8	23710	11603	11795	23518
10	23679	11634	11259	24054
20	23419	11894	2479	32834
40	22374	12939	2479	32834

Table 7.4: 5 workers, 150 rows, 80 columns: 12000 cells, 35313 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
4	215292	106641	236348	85585
8	215250	106683	107399	214534
10	215219	106714	105739	216194
20	214959	106974	21839	300094
40	213914	108019	21839	300094

Table 7.5: 5 workers, 450 rows, 240 columns: 108000 cells, 321933 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
8	8	22	8	22
15	8	22	8	22
20	8	22	8	22
40	8	22	8	22
80	8	22	8	22

Table 7.6: 10 workers, 6 rows, 3 columns: 18 cells, 30 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
8	157	137	157	137
15	64	230	64	230
20	22	272	22	272
40	22	272	22	272
80	22	272	22	272

Table 7.7: 10 workers, 15 rows, 8 columns: 120 cells, 294 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
8	2031	1005	1918	1118
15	1897	1139	918	2118
20	1740	1296	857	2179
40	670	2366	158	2878
80	68	2968	158	2878

Table 7.8: 10 workers, 45 rows, 24 columns: 1080 cells, 3036 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
8	23710	11603	24746	10567
15	23576	11737	11744	23569
20	23419	11894	10499	24814
40	22374	12939	1279	34034
80	18184	17129	1279	34034

Table 7.9: 10 workers, 150 rows, 80 columns: 12000 cells, 35313 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
8	215250	106683	225554	96379
15	215116	106817	107316	214617
20	214959	106974	103399	218534
40	213914	108019	11039	310894
80	209724	112209	11039	310894

Table 7.10: 10 workers, 450 rows, 240 columns: 108000 cells, 321933 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
48	8	22	8	22
90	8	22	8	22
120	8	22	8	22
240	8	22	8	22
480	8	22	8	22

Table 7.11: 60 workers, 6 rows, 3 columns: 18 cells, 30 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
48	22	272	22	272
90	22	272	22	272
120	22	272	22	272
240	22	272	22	272
480	22	272	22	272

Table 7.12: 60 workers, 15 rows, 8 columns: 120 cells, 294 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
48	111	2925	111	2925
90	68	2968	68	2968
120	68	2968	68	2968
240	68	2968	68	2968
480	68	2968	68	2968

Table 7.13: 60 workers, 45 rows, 24 columns: 1080 cells, 3036 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
48	21760	13553	18537	16776
90	9879	25434	8348	26965
120	8349	26964	6731	28582
240	229	35084	379	34934
480	229	35084	379	34934

Table 7.14: 60 workers, 150 rows, 80 columns: 12000 cells, 35313 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
48	213300	108633	214644	107289
90	208239	113694	105425	216508
120	202734	119199	81099	240834
240	164964	156969	2039	319894
480	1164	320769	2039	319894

Table 7.15: 60 workers, 450 rows, 240 columns: 108000 cells, 321933 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
80	8	22	8	22
150	8	22	8	22
200	8	22	8	22
400	8	22	8	22
800	8	22	8	22

Table 7.16: 100 workers, 6 rows, 3 columns: 18 cells, 30 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
80	22	272	22	272
150	22	272	22	272
200	22	272	22	272
400	22	272	22	272
800	22	272	22	272

Table 7.17: 100 workers, 15 rows, 8 columns: 120 cells, 294 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
80	68	2968	68	2968
150	68	2968	68	2968
200	68	2968	68	2968
400	68	2968	68	2968
800	68	2968	68	2968

Table 7.18: 10 workers, 45 rows, 24 columns: 1080 cells, 3036 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
80	18184	17129	18184	17129
150	6369	28944	6369	28944
200	229	35084	229	35084
400	229	35084	229	35084
800	229	35084	229	35084

Table 7.19: 100 workers, 150 rows, 80 columns: 12000 cells, 35313 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
80	209724	112209	183403	138530
150	195654	126279	88760	233173
200	180354	141579	84116	237817
400	67909	254024	1589	320344
800	689	321244	1589	320344

Table 7.20: 100 workers, 450 rows, 240 columns: 108000 cells, 321933 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
187	8	22	8	22
351	8	22	8	22
468	8	22	8	22
936	8	22	8	22
1872	8	22	8	22

Table 7.21: 234 workers, 6 rows, 3 columns: 18 cells, 30 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
187	22	272	22	272
351	22	272	22	272
468	22	272	22	272
936	22	272	22	272
1872	22	272	22	272

Table 7.22: 234 workers, 15 rows, 8 columns: 120 cells, 294 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
187	68	2968	68	2968
351	68	2968	68	2968
468	68	2968	68	2968
936	68	2968	68	2968
1872	68	2968	68	2968

Table 7.23: 234 workers, 45 rows, 24 columns: 1080 cells, 3036 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
187	229	35084	229	35084
351	229	35084	229	35084
468	229	35084	229	35084
936	229	35084	229	35084
1872	229	35084	229	35084

Table 7.24: 234 workers, 150 rows, 80 columns: 12000 cells, 35313 cache accesses

Cache Size	Diagonal		Vertical	
	Misses	Hits	Misses	Hits
187	184801	137132	183099	138834
351	76862	245071	76349	245584
468	53255	268678	52393	269540
936	689	321244	1139	320794
1872	689	321244	1105	320828

Table 7.25: 234 workers, 450 rows, 240 columns: 108000 cells, 321933 cache accesses

7.3 C: The BLOSUM62 Scoring Matrix

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1	-3	-1	-1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1	-3	-3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2	-2	-1	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2	0	0	0
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0	-1	-1	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3	-4	-3	-2
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1	-3	-2	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4	-3	-2	-1
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	-3	-3	-2	0	-1	-4	-3	-3	4	1	-1
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4	-1
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1