Lab on HW-SW digital systems codesign
SEL B.Sc. Curriculum (Spring Semester)
Embedded Systems Laboratory
EPFL-STI-IEL

Exercises Session 8
Out: 2025-04-10

Dynamic job scheduling across multiple accelerators

1 Introduction

The goal of this exercise is to explore different techniques to improve the performance of our accelerators by parallelizing the work over multiple units in the FPGA. As a target, we will consider the problem of **alignment of genomic sequences**.

1.1 Problem definition

"In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences." [1]

Sequence alignment is a critical tool in bioinformatics that allows researchers to study the similarity between genomes of different species or to trace possible evolutionary paths. When two sequences for the same protein are aligned, it may be possible to find out individual mutations between them, represented by nucleotide changes. Sequence alignment also exposes possible deletions or insertions of nucleotides in the sequence, by shifting one sequence with respect to the other. An intermediate point in sequence alignment is the determination of the similarity score between two sequences. In this project, we will focus on the generation of the similarity score.

Today, researchers count on many algorithms to perform sequence alignment. These algorithms are designed to be extremely efficient since the amount of data to be processed is often in the order of GiB or even TiB. Thus, execution time has always been an optimization target. However, recent concerns about energy consumption in data centers, both from the environmental and technological points of view, have motivated the switch from "time-to-completion" to "energy-to-completion."

In this exercise, we want to design a set of hardware accelerators that save time and energy while performing the alignment of a large number of sequences. To simplify, we will assume that the sequences of the database and of the specimen fit in the DRAM memory of the Pynq.

As a starting point, you will receive the original non-accelerated —pure software— version of the code. This version uses OpenMP to process the data using the two ARM cores of the Zynq FPGA at the same time. In addition, you will receive an initial basic implementation of the hardware accelerator, including both a complete Vitis HLS project, a Vivado project and the software application that uses this initial accelerator in the Pynq board. Unfortunately, the designers assigned to this project were fired due to budget cuts before finishing its optimization, so this version is much slower on the Pynq board than the original software version.

The file package includes also a "`Makefile`" to help in the development of the project, and an input dataset consisting of:

- A database with 40 000 genomic sequences.
- A file containing 1 000 sequences from a given specimen.
- A binary file containing the score of each comparison (i.e., the output of the algorithm). This is the ground truth for your implementation. To save space, we include an MD5 checksum of the file rather than the complete file.

The exercise consists on completing the source code for the two additional versions:

- A version that improves on the basic HW by caching the current database and specimen lines in an internal (BRAM) buffer.
- A version that uses HLS streams and tasks to distribute the comparison work across multiple workers in the FPGA, keeping the same interface with the SW.

1.2 Size of the problem and execution time

With the purpose of bounding the execution time of the tests, the input files supplied contain a database with 40 000 sequences and a specimen with 1 000 sequences. These sizes create the following execution scenario:

PARAMETER	VALUE		SIZE (B)
MAX_SEQ_LENGTH	32		32
DB entries	40 000	$40\,000 \times 32 =$	1 280 000
Specimen entries	1 000	$1\,000 \times 32 =$	32 000
DB lengths	40 000	$40\,000 \times 1 =$	40 000
Specimen lengths	1 000	$1\,000 \times 1 =$	1 000
Num of comparisons		$40\,000 \times 1\,000 =$	40 000 000
Scores		$40\,000\,000 \times 1 =$	40 000 000
Total RAM			41 353 000

The strings have a variable size of up to 32 nucleotides. Since they are stored in the files consecutively (i.e., according to their variable size), we need to create a list of string sizes, both for the database and for the specimen strings.

The following table shows the execution times and use of resources of each version:

NAME	Time (s)	LUTs	FFs	BRAMs	DSPs	Performance (comps/s)
00_SW	321.7 s	–	–	–	–	124 333
01_Basic_HW	5 192.0 s	3 380	4 113	1.5	4	7 705
03_CacheLines	379.6 s	3 470	4 299	1.5	4	105 362
05_Workers_x2	179.2 s	5 759	8 532	12	5	223 174
05_Workers_x4	90.4 s	8 293	12 426	15	7	442 441
05_Workers_x8	56.7 s	13 472	20 452	21	14	705 471
05_Workers_x10	54.4 s	16 100	24 495	14	13	734 630
05_Workers_x16	54.4 s	23 714	36 476	33	19	735 597
05_Workers_x24	54.4 s	34 007	52 208	45	27	735 727

2 Background: The Smith-Waterman algorithm for sequence alignment

The Smith-Waterman algorithm employs *dynamic programming* to find the optimal *local* alignment score of two genomic sequences. Additionally, it allows producing all optimal local alignments —we will skip this part in this exercise. In this context, local means that the sequences have already been shifted to match the correct comparison positions. Other algorithms, such as the one proposed by Needleman-Wunsch, can be used to compute global alignment. In that case, the reconstruction of the aligned sequences includes 'holes' that shift one sequence with respect to the other.

2.1 Step 1: Building the scores matrix

Given two sequences, A='ACCATGA' and B='ACCAGA', of lengths n and m, respectively, the algorithm starts to build a matrix M of size $(n+1) \times (m+1)$. Sequence A is written on top of the first row, whereas sequence B is written along the left border of the table. The first row and the first column of the table are filled with zeros:

← j →					$A[j]$				
			A	**C**	**C**	**A**	**T**	**G**	**A**
↑		0	0	0	0	0	0	0	0
i	**A**	0							
↓	**C**	0							
	C	0							
$B[i]$	**A**	0							
	G	0							
	A	0							

The table is filled from top to bottom, and from left to right, using the following formula:

$$M[i,j] = \max \begin{cases} 0 \\ M[i-1,j-1] + S(A[j],B[i]) \\ M[i-1,j] - 1 \\ M[i,j-1] - 1 \end{cases} \tag{1}$$

where:

$$S[a,b] = \begin{cases} +1, & \text{if } a = b \\ -1, & \text{if } a \neq b \end{cases} \tag{2}$$

Applying formula 1, we can start filling in the first row, from left to right.

← j →					$A[j]$				
			A	**C**	**C**	**A**	**T**	**G**	**A**
↑		0	0	0	0	0	0	0	0
i	**A**	0	1	0	0	1	0	0	1
↓	**C**	0							
	C	0							
$B[i]$	**A**	0							
	G	0							
	A	0							

The final configuration of the table is as follows:

					(A)				
			A	**C**	**C**	**A**	**T**	**G**	**A**
		0	0	0	0	0	0	0	0
	A	0	1	0	0	1	0	0	1
	C	0	0	2	1	0	0	0	0
(B)	**C**	0	0	1	3	2	1	0	0
	A	0	1	0	2	4	3	2	1
	G	0	0	0	1	3	3	4	3
	A	0	1	0	0	2	2	3	5

The algorithm has to track the maximum value in the table — we will assume in this example that we keep the first appearance of this maximum value, in the order in which the values are calculated:

(A)

			A	C	C	A	T	G	A
		0	0	0	0	0	0	0	0
	A	0	1	0	0	1	0	0	1
	C	0	0	2	1	0	0	0	0
(B)	C	0	0	1	3	2	1	0	0
	A	0	1	0	2	4	3	2	1
	G	0	0	0	1	3	3	4	3
	A	0	1	0	0	2	2	3	5

In this simplified version of the problem, the result ("score") is the maximum value found while building the table.

3 Additional instructions

3.1 Structure of the provided project

The project provided contains several folders:

- "`00_SeqMatcher_SW_OpenMP/`" Original implementation of the sequences in SW using OpenMP. Works in any Linux machine. Change the number of threads in the Makefile to the number of cores in your machine. The Pynq board has two ARM cores.
- "`01_SeqMatcher_HW_Basic/`" Baseline implementation of the HW accelerator. Contains the Vitis HLS, Vivado and Pynq Linux (SW) projects. Use the Makefile inside to generate the different parts of the project.
- "`03_SeqMatcher_HW_Basic_CacheLines/`" Baseline HW version with caching of the current line from the database and from the specimen in an internal BRAM buffer. This version has to be completed.
- "`05_SeqMatcher_HW_Workers/`" HW version with HLS streams and tasks to distribute the comparisons across multiple workers inside the FPGA. This version has to be completed.
- "`testdata/`" Contains the main database and specimen files, with the scores output generated with the original SW version.

3.2 Using the project "`Makefile`"

The included "`Makefile`" contains the following targets:

- Vitis HLS targets

  hls_project: Just creates the Vitis HLS project
  hls_sim: Creates the Vitis HLS project and runs the C++ simulation
  ip: Creates the Vitis HLS project, synthesizes the design and exports the IP core

- Vivado targets

  vivado_project: Just creates the Vivado project
  bitstream: Creates the Vivado project and runs synthesis up to bitstream generation
  extract_bitstream: If the Vivado bitstream has already been generated, extracts it to this folder

- Generic targets

  clean: Deletes log files and Vitis HLS and Vivado projects; deletes the files in the IP catalog, but keeps the IP catalog ZIP file
  cleanall: Additionally, deletes the bitstream files and the IP catalog ZIP file
  help: Displays a short help message

The project can be recreated from scratch using the *bitstream* target. If previous steps are already completed, `make` will construct only the subsequent steps.

Inside the `SW_` folder for the Pynq, there is another `Makefile` intended to build the software in the Pynq board. In this case, the flag `-j2` is passed to `make` to parallelize the compilation across the two ARM cores of the board, which saves some time.

3.3 File formats

Each sequence in the database and specimen files is stored as an ASCII string with letters representing the nucleotides ('A', 'C', 'G', 'T'). Each sequence can have a length between 16 and 32 nucleotides, both inclusive. In the files, each sequence terminates with a new-line character. The strings are loaded into memory in consecutive form, that is, without line breaks or `NULL` markers. Additionally, to save memory, the sequences are stored in memory with their actual length (i.e., the application does not reserve the maximum space for each string).

Therefore, the application creates two additional vectors to store the lengths of each of the strings: one for the database entries and one for the specimen entries, respectively. Since the length of each sequence is limited to a maximum of 32 nucleotides, the lengths are stored as `uint8_t` values. For example, the following data in a file:

```
ACCGACGTGACGTACTTC
GACGTGACGTACTTGA
```

would be represented in memory as: "ACCGACGTGACGTACTTCGACGTGACGTACTTGA" and the vector of lengths would contain the values: {18, 16}. The accelerator implementation *must* respect the format of the files, but it *is free* to modify the representation of the sequences in memory.

Scores are usually limited to small positive or negative values. In this example, we will represent them as `int8_t` values. The scores are stored consecutively in the file. The accelerator implementation *must* respect the format of the output file.

3.4 Faster execution during debugging

The database file can be shortened to reduce execution times, particularly while working with the initial (slower) versions of the accelerator. To produce a golden reference, execute the original SW version indicating a smaller number of entries in the DB. Save the scores file with a different name and use that file as a reference to compare the correctness of your implementations during debugging.

4 Acknowledgments

We would like to acknowledge the help of María Elena Espinosa, from the University of Málaga, for introducing us to the challenges posed by the problem of sequence alignment during her Ph.D. stay at ESL-EPFL, and for providing hints in the choice of algorithms for this exercise.

References

[1] Wikipedia. Sequence alignment. [Online]. Available: https://en.wikipedia.org/wiki/Sequence_alignment