Performance of Smith-Waterman DNA sequence matching on a FPGA

1 Juli 2022

P. Kingma
p.j.kingma@student.utwente.nl
University of Twente, Enschede

***Abstract*—Because of advancements in molecular sequencing technology, a need for even faster processing of this fast-growing pool of data is needed. Preferably, this data is processed as soon as possible, as fast as possible, so having a low latency and high throughput is of importance. This usually includes finding matches between certain DNA sequences. These algorithms have a lot of parallelizable operations, and could have significant improvements in latency and throughput when implemented on a FPGA. One such algorithm is Smith-Waterman, which locates the best local alignment according to the set scoring system. The goal of this paper is to explore what the limits of current implementations of Smith-Waterman are, and how these limits show in the performance metrics of the algorithm's hardware implementation.**

CONTENTS

1. INTRODUCTION

The usage of FPGA's (Field Programmable Gate Array) to accelerate algorithms is not new to the medical field [1]. With the fast amount of data that the field generates and needs processing for in mind, it is logical that these algorithms are among the first to give use to what FPGA's offer in terms of computational power. One such algorithm is Smith-Waterman, designed to find the best local alignment (partial match) of two sequences, in this case DNA sequences. These algorithms, including Smith-Waterman, are still being improved. As becomes apparent with current state of the art implementations offering throughputs up to 214 GCUPS (Giga Cell Updates per Second) [2]. These implementations offer tremendous performance.

To get an overview of how what limits the performance of Smith-Waterman hardware implementations, as well as trade-offs between performance, resource usage, power usage and cost, a broader look is needed. Implementing a parametric design for Smith-Waterman allows to keep the same architecture, and explore the effect of only changing one parameter. A look at these effects can help in deciding what trade-offs to make for certain use cases.

This paper discusses what parameters currently limit implementations of Smith-Waterman in their performance and resource usage. This information can then be used to estimate FPGA requirements and/or as a start for future research, showing what change offers what improvement. To attempt this, the same High Level Smith-Waterman design is synthesised while only changing one parameter at a time.

But first, the background to this paper is explained, followed by a discussion and comparison with related work. Then the design is presented, divided into architecture and implementation. Finally, the results are presented, discussed and concluded in the conclusion at the end.

2. BACKGROUND

A. Smith-Waterman

The Smith-Waterman algorithm was first published in 1981 by T.F. Smith and M.S. Waterman [3]. The algorithm finds the mathematical best local sequence alignment, that is, it determines regions of similarity following a certain scoring system. The algorithm is a variation on Needleman-Wunsch [4], by setting negative scoring cells to zero, a positively scoring local alignment becomes visible in the scoring matrix. This ability to find local alignment instead of global alignment is Smith-Waterman's main difference compared to needleman-Wunsch.

The scoring system consists of a certain score for two matching sequence elements, and a gap penalty when a gap is inserted. These can be constant, but also implemented as a linear gap penalty for example, giving a higher gap penalty as the inserted gap gets larger. Also the score for matching two sequence

elements can be dependant on what sequence element it is, although this is not used very often, as generally any match is worth the same.
Now a score matrix can be constructed, with one of the sequences along the horizontal, and one along the vertical axis of the matrix. The first column and row are initialized to zero, see figure 1. With a chosen scoring system in hand, the scoring

		T	G	T	T	A	C
	0	0	0	0	0	0	0
G	0	0	3	1	0	0	0
G	0	0	3	1	0	0	0
T	0	3	1	6	4	2	0
T	0	3	1	4	9	7	5
G	0	1	6	4	7	6	4
A	0	0	4	3	5	10	8

Fig. 1: An example score matrix, based on [5]

can begin. For every cell, three scores will be evaluated:

- The score to the left, minus the gap penalty
- The score to the top, minus the gap penalty
- The score to the top-left, plus or minus the score for match depending on if the words are a match

If the highest score of the three is positive, it will be inserted in the matrix, otherwise a zero is inserted. And this goes on for the whole matrix until completion. Or mathematically, this can be put in eq. 1.

$$h_{i,j} = max \begin{cases} h_{i-1,j} - gapPenalty \\ h_{i,j-1} - gapPenalty \\ h_{i-1,j-1} \pm matchScore & \text{Sign dependent on match or not} \\ 0 \end{cases} \tag{1}$$

Now that the score matrix is determined, the trace-back step can start. This involves finding the highest score, and then following back from which surrounding cell it got its score. This process continues until a cell with score zero is found. This trace-back now creates the part of the reference and the query, with insertions, that match mathematically optimal, following the chosen scoring system.

B. Parallelism and Sequentialism in Smith-Waterman

The Smith-Waterman scoring system is based on scores to the top, to the top-left and to the left of the cell that is being scored. This dependency makes for the possibility that the anti-diagonal

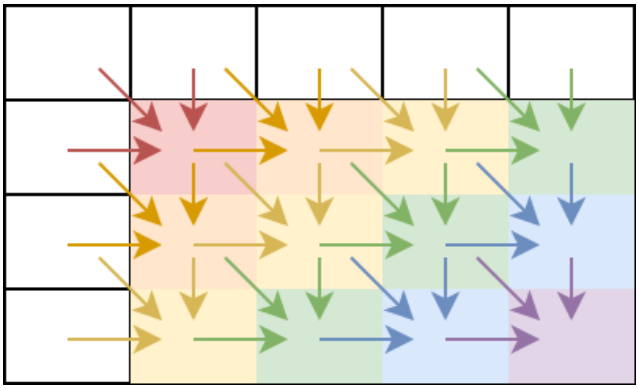

Fig. 2: Parallelism of Smith-Waterman, the colored arrows show what data dependency a cell has, the cell color shows which cells can be calculated simultaneously, without being dependent on each other

of the scoring matrix can be determined at once, without data dependency conflicts. This can be seen in an example part of a score matrix in figure 2.
When applying this parallelism to hardware design, this keeps the LUTs (Look Up Table) smaller, as a certain score is only dependant on a certain part of the input. This reduces the amount of inputs the LUT needs, reducing the resource requirement of the LUT, and allowing higher clock frequency as critical paths are shorter. It is important to make the synthesis tool aware of this parallelism in the anti-diagonal, as otherwise it will go through each row, left to right, requiring larger LUTs as the inputs required are larger.
It must also be noted that the length of a row will be related to the latency, as the inherent sequential calculations create a critical path that can be pipelined, increasing throughput, but can't be shortened, increasing the latency as the reference sequence increases.

C. FPGA

A FPGA (Field Programmable Gate Array) is an integrated circuit designed to be (re)configured after manufacturing. A design on an FPGA is often described in a Hardware Description Language (HDL). This versatility to be configured makes a FPGA a great tool to test ASIC (Application Specific Integrated Circuit) designs, as ASIC's are often described in HDL as well. Another application of a FPGA is as a hardware accelerator besides a "regular" processor. In this case, when a user starts a workload that could benefit from the parallelised processing a FPGA offers, the corresponding HDL design is implemented on the FPGA, and the processor offloads the workload to the FPGA.
This application of a FPGA is what is used in this paper, as Smith-Waterman becomes a very large workload when taking into account the scale of DNA sequences.

3. Related Work

As the Smith-Waterman algorithm [3] is commonplace in the medical data-analysis field, the algorithm has seen it's share of implementations and improvements, on both CPU's and GPU's, as well as hardware accelerators like FPGA's. The work by [1] presents a comparison of the performance diffences between mentioned hardware. GPU's show an eight-fold improvement in throughput over CPU implementations, while FPGA implementations show improvements from 6- to 24-times compared to CPU implementations [1]. This shows that Smith-Waterman as an algorithm can make great use of parallelism, as more parallelisation focussed hardware like GPU's and FPGA's show a great improvement.

Another aspect is power usage, both from a costs perspective as well as a space perspective. FPGA's offer an improvement from 8- to 66-times in power usage when compared to GPU implementations [1]. This shows the advantage an FPGA has over a GPU, saving power cost, and being able to be packed denser together, because of less cooling needed when compared to a GPU.

The currently highest performing Smith-Waterman implementation by throughput is Houtgast [2]. This paper shows that improvements could be made by making sure all hardware is also always "busy". Their implementation uses 99.8% of the hardware continuously, while their predecessor in throughput, Sirasao [6] "only" used 56.9% of hardware continuously. This difference in hardware usage contributes to their three-fold improvement over Sirasao [2]. It must also be noted that Houtgast used an Intel Arria 10 GX FPGA Development Kit, but also presented the performance of their implementation on an FPGA with a lower fabric speed grade. Here, the importance of the used hardware becomes apparent, as their implementation improved linearly with the higher frequency they were able to run on the Arria 10 Dev kit.

To compare latency performance is more difficult, as different implementations can have different definitions of latency. In this paper, it is defined as the time between having received a set of words from the query stream, and having calculated the corresponding rows. To determine this latency from related work, eq. 2 is defined. Although this equation misses start-up and finishing time of the rows, that measurement is seldom presented in papers, so it would be difficult to obtain. It can also be assumed that start-up and finishing times are less significant to the calculation of the whole row, because loading M words should only take one cycle.

$$L_{packet} = N_{cells\,per\,row} + M_{words\,per\,set} - 1 * T_{cycle\,period} \tag{2}$$

This definition of latency is chosen as it still includes the parallelism from section 2-B, because of M.

As for latency performance, a current implementation that also targeted latency is de Oliveira [7]. This paper offers a really specific look at how each step of Smith-Waterman can be implemented, and explained on the level of the individual logic elements. The suggested implementation has as main advantages a reduced hardware resource usage on the used FPGA, while maintaining a high performance. Besides showing an implementation, the paper also clearly shows how they measured the performance of their implementation, which provides a solid reference when comparing performance.

From table I the performances of the related works, as well as this paper's performance can be compared.

As the main source to implement a parametric Smith-Waterman implementation, the work in [8] must be mentioned. It identifies how Smith-Waterman can be implemented in hardware, and what optimization strategies can be used to exploit the parallelism in Smith-Waterman, which has been detrimental to the implementation designed in this paper.

4. Architecture

The architecture first covers a few terms that are important to be understood.

word	Single sequence element, in case of DNA, an A, C, T or G
Reference sequence	The sequence of words completely loaded into memory at the beginning
Query sequence	The sequence of which an amount of words set by a parameter is streamed to the architecture every clock cycle

TABLE II: Often used terms

First, the inputs and outputs are discussed, as they are what the master device sees when interfacing with the architecture. Afterwards, the internals of the architecture are discussed individually.

A. IO

The architecture features three IO ports, two inputs streams, one for the reference and one for the query, and an output stream, which outputs the highest score and its location in the scoring matrix continuously.

To the device interfacing with the architecture, there are three steps:

- Loading the reference sequence into memory
- Streaming the query sequence in the FPGA
- Streaming the highest score out of the FPGA

Step two and three happen continuously, as for every part of the query sequence streamed to the FPGA, an highest score output is returned.

1) Loading the reference sequence into memory: This step can be seen as the start-up step, as first a reference sequence is loaded into memory. By only loading one sequence into memory, the resource requirement of the architecture is kept low. Streaming the query sequence also allows for improved throughput, as the architecture is already processing the first words, and doesn't have to wait until the whole query is loaded in.

	GCUPS	Frequency (MHz)	Latency (ns)	Resource Usage (LUTs)
Houtgast [2] *1	214.8	164	853.6	1046500
Houtgast [2] *2	107.4	137	1022	678500
de Oliveira	79.5	155	1094	35286
This paper	115.68	225.94	1341	60995

TABLE I: Performance of discussed papers. *1 is the maximum performance presented, *2 is performance on the lower fabric speed FPGA.

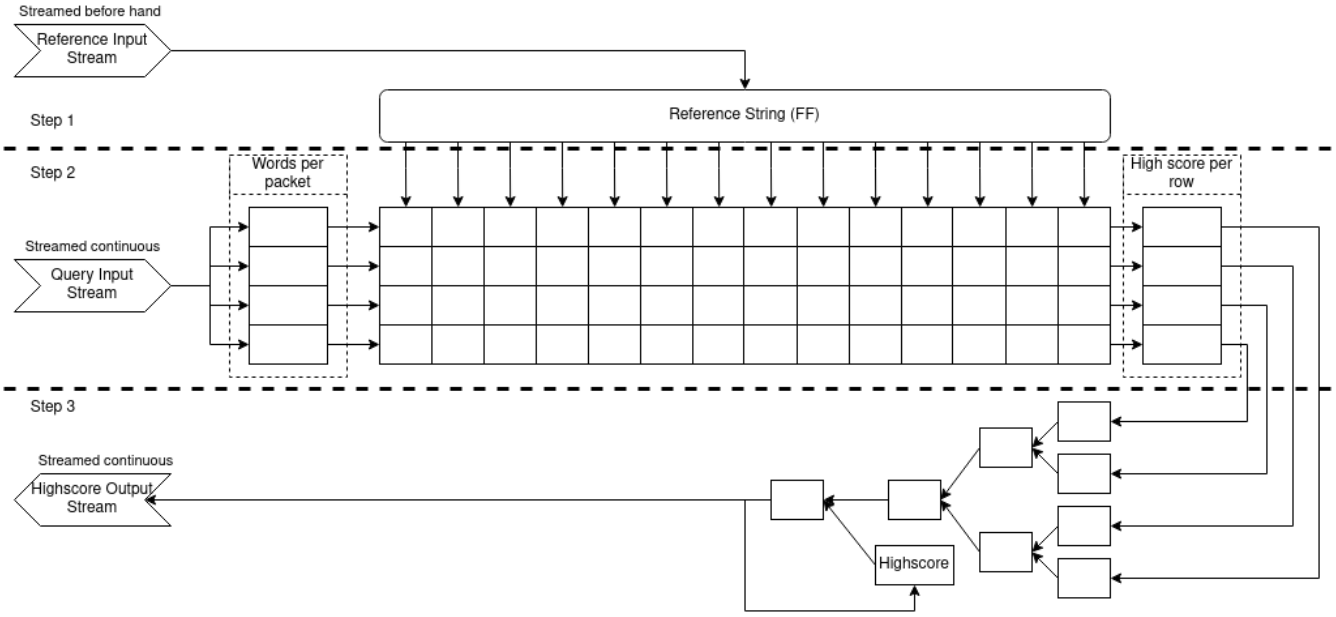

Fig. 3: The hardware architecture, note the three steps mentioned in 4-A.

2) Streaming the query sequence to the FPGA: On the query sequence input, a certain amount of words are received. These amount of words dictate the amount of rows that can be determined simultaneously improving throughput following section 2-B. Using this, a two-dimensional array is initialized, sized adequately to the length of the reference sequence and the width of the amount of words. In this array the scores are calculated, and the highest score of each row is kept track of, to be output later. The highscores are kept per row, as rows are inherently sequential, keeping track of the highest score does not introduce data dependency issues, while keeping track of the highest score in all rows at the same time does. To find the total highest score, a separate segment of hardware is used.

3) Streaming the highest score out of the FPGA: To efficiently determine the highest number of the simultaneous determined highscores of the rows, they are compared one-by-one, creating a tree-like structure as can be seen in the bottom right of figure 3. This part receives the stream of highscores per row, and starts finding the highest score. This highest score is then finally compared to the highest score until then, and the highest is output in the end, along with it's row and column.

B. Score Calculation

The score calculation of a single cell follows eq. 1 strictly. The three scores are determined, and compared for the largest (also with zero). Then the largest is stored in the cell.

C. Rows Calculation

As multiple words arrive per clock cycle on the query stream, multiple rows can be calculated using the parallelism discussed in section 2-B. This is implemented by two for loops, one nested inside the other, with the upper loop going through the words received on the stream, and the nested loop going through each word on the reference query. To improve latency, these loops are unrolled, as by default nested loops are implemented taking a clock cycle to change from the upper to the nested loop. The latency between starting in the upper left and finishing the final cell in the bottom right of figure 2 takes, assuming the goal from 2-B to calculate one anti-diagonal every clock cycle:

$$Latency_{Rows} = (N_{reference} + N_{Wordsperclockcycle} - 1) + 1 \tag{3}$$

This relation becomes apparent when looking at the shortest path between the first cell (left-top) and the final cell (bottom-right). Because an extra clock cycle is needed to add the highscore of

the row to the output stream, an extra clock cycle is added to the latency to account for this.
Because the throughput of at maximum only calculating one anti-diagonal per clock cycle is rather low, and makes low use of the allocated resources, this row calculating function is pipelined with no delay, so every clock cycle as many rows are output as the query stream inputs words.

D. Upper functionality

The upper functionality is basically the connection between the streams on the IO, and the internals. The upper functionality first streams the the reference into memory. And then proceeds read the query stream every clock cycle, and putting the received query sequence into the rows calculation pipeline. It also connects the highscores per row stream to the comparator to find the highest score, and output it back to the master device.

5. Implementation

A. High Level Synthesis

The architecture is implemented in Vitis High Level Synthesis version 2022.1 build 3526262 [9], running on Ubuntu 20.04.4 LTS. The computer synthesizing the design has an AMD Ryzen 7 3700x CPU, with 16 GB of memory, along with a swapfile of 64 GB to work with the larger designs.
In the final design, only five parameters have to be set for an accordingly scaled architecture:

- The word size (bits)
- The number of words in per packet on the input stream (Number)
- The number of words in the reference query (Number)
- The gap penalty (Integer)
- The match score (Integer)

This parametric design allows us to generate design points of varying size and performance, allowing for adaptation to different FPGA's, as well as re-configuring the architecture to use a larger reference sequence and less words per packet or vice versa as needed. As all the derivative parameters are determined at compile time, they appear as constant expressions to the synthesis tool, allowing for the complete unrolling of for loops. For example, the target latency to calculate a row following eq. 3 is implemented as a Vitis HLS pragma (#pragma HLS latency max=), in which this max latency is determined by a constant expression function at compile time.
1) Loading the reference sequence into memory: To load the reference sequence into memory, an HLS STREAM is used, as per the examples [10]. This memory is fully parallel, meaning all elements can be accessed every clock cycle, to allow for the pipelining of the score matrix calculations.
2) Streaming the query sequence to the FPGA: As the calculation of the scores is implemented as two nested for loops, a loop for each word on the input, and a nested loop for each reference element. This unrolling is necessary to parallelize as much operations as possible, as by default, entering and exiting a nested loop both cost a clock cycle. Besides this, by unrolling the synthesis tool explores what can be executed concurrently, and in combination with setting a certain latency for this to run, will realize the anti-diagonal parallelism from section 2-B. An attempted implementation had the for loops designed to go through the anti-diagonal, but this stopped the loops from being completely unrolled, as the nested loop's iteration was dependent on the outer loop.
3) Streaming the highest score out of the FPGA: To create the tree-structure to determine the highest score, two nested for loops are used, an outer loop that goes through the levels of the tree, an a nested loop going through the individual comparisons. As the amount of comparisons is related to the level of the tree by $2^{treelevel}$, the implementation has to take powers of 2. Raising a number to a certain power is costly in hardware, but a power of two is essentially a left shift of a binary one by said power. This operation is much easier to do in hardware, and saves resources.

B. Limitations of the implementation

This implementation of Smith-Waterman does not allow for a match score matrix, which means giving different scores for different matches, and also does not allow for a non-linear gap penalty systems. Also, the output is only the row, column and score of the highest cell in the score matrix at that time. This means the host system will have to do the trace-back via a reverse calculation of the three scores, moving to the correct one, and repeat this until finding a zero, which is costly.

C. Test bench

A test bench was designed to validate the design. The test bench creates a random reference and query sequence, and inputs it to the simulated FPGA. When the FPGA simulation is complete, the same reference and query sequence are put through a validated C++ Smith-Waterman implementation. Then the outputs of the FPGA simulation and the C++ implementation are output to the user after running the test bench, and it can then be verified.
To also test if the synthesized implementation works, a co-simulation is run with the same test bench. The co-simulation is then verified by the timing of the synthesized modules of the implementation, as well as general output correctness.
The test bench, as well as the implementation can be found on gitlab [11].

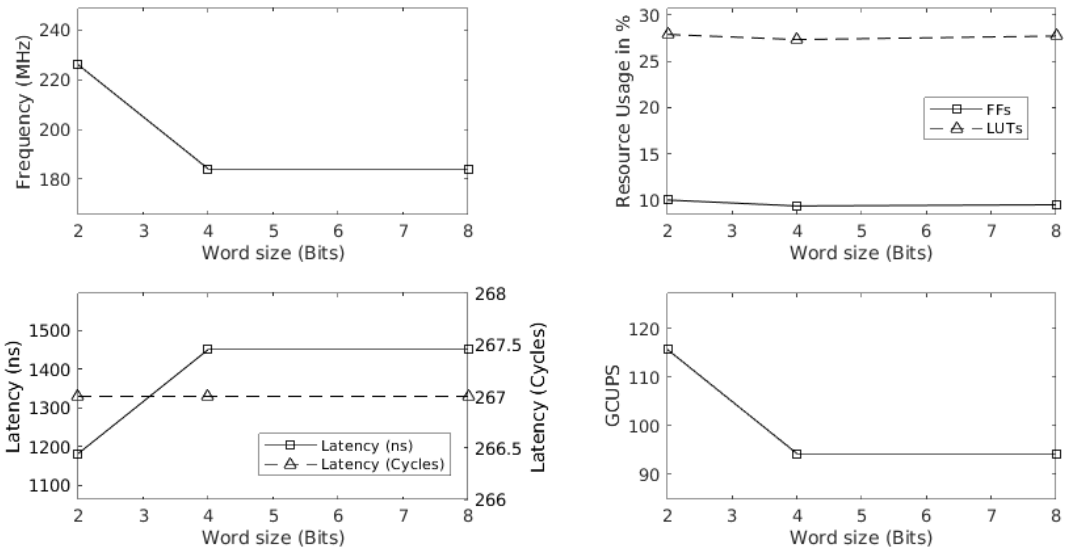

Fig. 4: The wordSize in bits, with a reference size of 128, and 4 rows in parallel

6. Results

These results are from synthesis and simulation in Vitis HLS, targeting the ZYNQ-7 ZC706 Evaluation Board, which uses the xc7z045ffg900-2 FPGA. To quantify how the three parameters (word size, rows calculated in parallel and reference size) influence the scale and performance of the architecture, the following three test cases were designed:

- Word size: 2, 4, 8; With a reference size of 128 and 4 rows calculated in parallel.
- Rows calculated in parallel: 1, 2, 4, 8; With a reference size of 128 and a wordsize of 2.
- Reference size: 64, 128, 256; With 4 rows calculated in parallel and a wordsize of 2.

A. Word size

The results of the variable word size can be seen in figure 4. Of interest is the lack of change in resource usage, frequency, latency and throughput when doubling the word size from four to eight bits, while the frequency and GCUPS did change significantly for two bits to four bits. This can be due to the synthesis tool prioritizing latency in cycles and resource usage over the frequency.

B. Rows in parallel

The results of the variable amount of rows calculated in parallel can be seen in figure 5. Of interest here is the linear increase in resource usage and GCUPS, indicating that increasing the amount of rows in parallel indeed exploits the parallelism from section 2-B. This also shows in the latency in cycles, as the cycles do not double when doubling the rows in parallel.

C. Reference size

The results of the variable reference sequence size can be seen in figure 6. Of interest is how the resource usage increases in similar fashion to the variable rows in parallel. What differs when comparing to the variable rows in parallel, is the drop in frequency, and doubling of the latency in ns when doubling the reference size. This does make sense however, as following eq. 3, with $N_{reference}$ being a much larger number than $N_{words\,per\,clock\,cycle}$, the doubling of $N_{reference}$ will also roughly double the latency, which is clearly visible. Also, the final result (reference size of 256) must be discussed, as it does not follow the rest of the results, and was due to the synthesis tool not successfully pipelining the row calculation function, causing the drop in frequency and especially GCUPS.

D. Discussion

As High Level Synthesis will always be a hardware interpretation of the provided high level code, results will vary between compiler/synthesizer settings, FPGA choice and optimization settings. So these results are only indicative when using another FPGA or settings, and will not compare well with results that change the FPGA or settings when changing parameters as well.
Also, the synthesis tool will make sacrifices when certain requirements are close to or over a certain limit of the chosen FPGA, in which case it will drop certain requirements, changing the architecture and influencing the performance metrics, which is what might be the reason of the deviation of the final result for the variable reference size in figure 6.

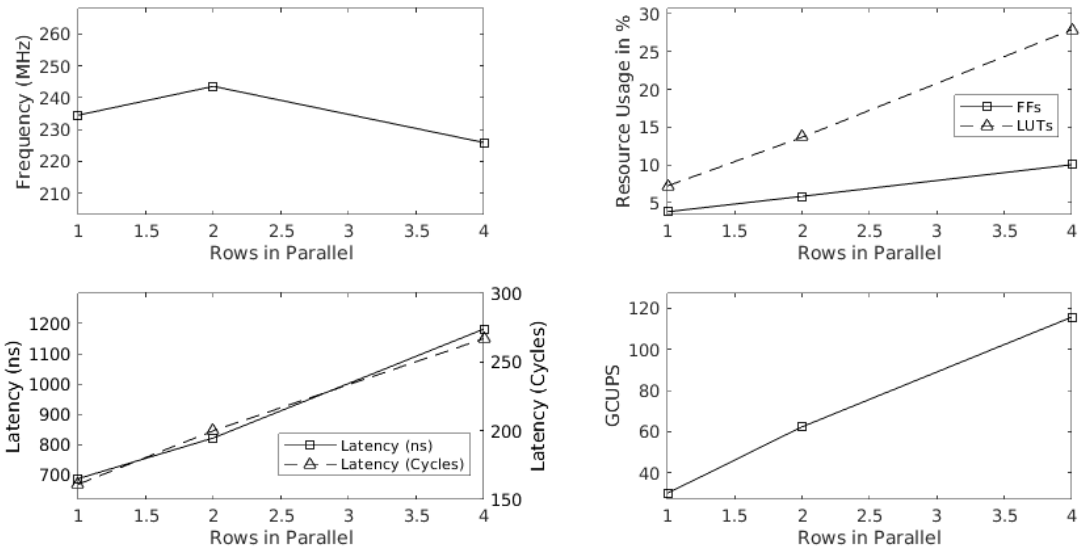

Fig. 5: The amount of rows in parallel, with a reference size of 128, and a word size of 2

7. Conclusion

The quantization of reference sizes, word sizes and bus widths on the latency, throughput and resource usage of a general Smith-Waterman implementation presented in this paper provide insight in the performance of running the implementation at a certain scale, and what can be expected when changing such parameters. The presented Smith-Waterman implementation obtained from a high level description is verified for correct performance using co-simulation and comparing the output with a verified software Smith-Waterman implementation.

With the focus of the paper being a parametric design and it's scalability, some losses are expected when compared to state of the art implementations. This becomes apparent when looking at table I, as this paper's implementation does not lead in any performance metric, but does stay in the ball park.

Concluding what parameter is worth further research is the increase in bus width, as this leverages the parallelism discussed in section 2-B.

8. Recommendations

As for future research, it is first of all important to run the same experiments on a physical FPGA, instead of simulation, as this would provide real world performance, instead of the theoretical maximum. This would also be a more fair comparison with other work, as theoretical performance is usually the same as or better than real-life.

The design could also allow for giving different scores to different matches, as well as a variable gap penalty, as this would make this implementation closer to the complete Smith-Waterman algorithm. This is shortly mentioned in section 5-B. Although it must also be noted that this would introduce another variable, as the complexity of both the variable match score as the variable gap penalty will add to the complexity of calculating a score.

In future implementations, the whole matrix could be outputted to the parent system, as that would speed up the trace-back stage of Smith-Waterman. In the current implementation, the host system has to reverse calculate the left, upper-left and upper score of every cell passed in the trace-back, instead of just finding the highest of the three. Another option is to integrate the trace-back into the design, but this would increase resource usage, as the complete score matrix has to be stored for the trace-back stage.

As synthesis times went up to twelve hours for the larger experiments, and memory usage to 60 Gigabyte, a more dedicated machine than the used computer are recommended, especially the memory, as working with a swapfile decreases a computer's performance significantly.

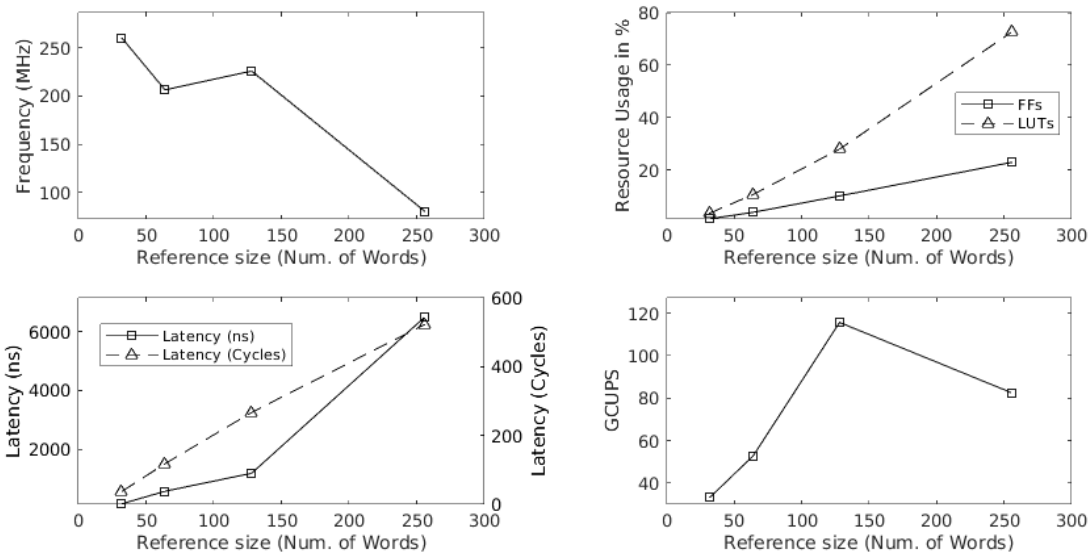

Fig. 6: The size of the reference sequence in words, with 4 rows in parallel, and a word size of 2

REFERENCES

[1] L. S. M. Bragança, A. D. Souza, R. A. S. Braga, M. A. M. Suriani, and R. M. C. Dias, "Sequence alignment algorithms in hardware implementation: A systematic mapping of the literature," *Advances in Intelligent Systems and Computing*, vol. 1346, no. 6, pp. 307–312, 2021.

[2] E. Houtgast, V.-M. Sima, and Z. Al-Ars, "High performance streaming smith-waterman implementation with implicit synchronization on intel fpga using opencl," in *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, 2017, pp. 492–496.

[3] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, 1981. [Online]. Available: http://dornsife.usc.edu/assets/sites/516/docs/papers/msw_papers/msw-042.pdf

[4] S. B. N. C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, 1970.

[5] "Smith-waterman algorithm." [Online]. Available: https://en.wikipedia.org/wiki/Smith-Waterman_algorithm

[6] R. S. A. Sirasao, E. Delaye and S. Neuendorffer, "Fpga based opencl acceleration of genome sequencing software," in *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, vol. 128, no. 8.7, 2015, p. 11.

[7] de Oliveira, F. F., Dias, L. A., Fernandes, and M. A. C., "Parallel implementation of smith-waterman algorithm on fpga," *bioRxiv*, 2021. [Online]. Available: https://www.biorxiv.org/content/early/2021/07/27/2021.07.27.454006

[8] X. Chang, F. A. Escobar, C. Valderrama, and V. Robert, "Optimization strategies for smith-waterman algorithm on fpga platform," in *2014 International Conference on Computational Science and Computational Intelligence*, vol. 1, 2014, pp. 9–14.

[9] "Xilinx Downloads." [Online]. Available: https://www.xilinx.com/support/download.html

[10] "Vitis HLS Introductory Examples." [Online]. Available: https://github.com/Xilinx/Vitis-HLS-Introductory-Examples

[11] "Fpga smith-waterman." [Online]. Available: https://gitlab.com/kingma1999/fpga-sw