Low-Power Vectorial VLIW Architecture for Maximum Parallelism Exploitation of Dynamic Programming Algorithms

Miguel Cruz, Pedro Tomás, Nuno Roma
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
e-mail: miguel.tairum@tecnico.ulisboa.pt, pedro.tomas@inesc-id.pt, Nuno.Roma@inesc-id.pt

***Abstract*—Dynamic Programming algorithms are widely used in many areas, to divide a complex problem into several simpler sub-problems, with many dependencies. Typical approaches explore data level parallelism by relying on spacialized vector instructions. However, the fully-parallelizable scheme is often not compliant with the memory organization of general purpose processors, leading to a less optimal parallelism, with worse performance. The proposed architecture exploits both data and instruction level parallelism, by statically scheduling a bundle of instructions to several different vector execution units. This achieves better performance than vector-only architectures, and has lower hardware requirements and thus lower power consumption. Performance and energy efficiency metrics were used to benchmark the proposed architecture against a dual issue, out-of-order ARM Cortex-A9 and a dedicated ASIP architecture. In a fair comparison where all processors compute 16 dynamic programming cells in parallel, results show that the proposed architecture can achieve a 3.24x and 2.35x better performance-energy efficiency than the ARM Cortex-A9 and the dedicated ASIP, respectively, and a performance improvement of 2.54x and 5.01x regarding the ARM and the dedicated ASIP, respectively.**

***Keywords*—Multiple Instruction Multiple Data Architecture, Dynamic Programming, VLIW, low-power, Instruction Level Parallelism, Data Level Parallelism**

I. Introduction

Dynamic Programming (DP) is a common methodology for solving complex problems, by dividing them into smaller sub-problems that are simpler to solve. This results in a sequence of sub-problems or states that depend on each other, with the latter states depending on the previous ones.

DP has been extensively applied in a vast set of different application domains. As an example, in the bioinformatics domain, DP is frequently used by sequence alignment algorithms like, the Needleman-Wunsch [1] and the Smith-Waterman [2], to align large sequences of DNA or protein elements by filling up a score matrix, where the computation of each cell depends on the upper, left and upper-left cells. Each cell can then be seen as a sub-problem that has only three previous cells as its dependencies, instead of all the previous computed cells. This property enables the use of DP methods to improve the performance of the algorithms. Another common application of DP is found in Hidden Markov Models, where the Viterbi algorithm [3] is frequently implemented by using DP methods. Starting with a sequence of observations for a given state space and the corresponding transition and occurrence probabilities, this algorithm finds the most probable state sequence that originated such observations. Each state thus only depends on the previous state and on the probabilities that lead to it, corresponding each state to a sub-problem of the algorithm. Therefore, DP methods are used to concurrently compute different states in the same iteration, increasing the performance of the algorithm. It is also important to notice that DP applications tend to be very computation-heavy due to their large datasets, and often require high performance in a low power environment, leading to the need of finding better solutions with lower energy budgets.

Typical processing solutions to solve this problem include General Purpose Processors (GPPs) [4], [5] and dedicated hardware [6], by using Single Instruction Multiple Data (SIMD) extensions. These extensions exploit Data Level Parallelism (DLP) by computing in parallel a vector composed of multiple cells, where each cell is often independent of all the others. Since DP algorithms work with large data banks, this type of parallelism is essential to obtain a good processing performance. Also, the amount of memory that is required to accommodate the dependencies between states on DP algorithms is often very large, requiring the implementation of techniques to cache and reuse results of previous state computations, in order to quickly retrieve them without redundant computations.

However, the DLP that is exploited in the existing solutions comes at a cost of high hardware requirements. Since each cell in the vector is computing the same instruction, the number of required functional units (FUs) will be proportional to the vector length, while the usage of those units will be low, given that only a small number of FUs related to the instruction being computed will be active at any given time. Also, for the GPP solutions, the memory organization is not particularly suited for some types of parallelism. Taking the Smith-Waterman algorithm as an example, each cell has three dependencies (as previously mentioned, the upper, left and upper-left cells). Hence, the only way to maximize the exploited parallelism is to process cells along the anti-diagonal of the matrix. However, this is not feasible in conventional GPP implementations, because of the resulting memory access pattern. To circumvent this issue, state-of-art implementations [4], [7] operate either horizontally or vertically, which improves the overall algorithm performance by taking advantage of a traditional GPP memory organization, but with the cost of additional lazy loops to solve the existing dependencies.

To overcome the previous issues, this paper proposes an innovative architecture that extends the conventional paradigms

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects: HELIX (PTDC/EEA-ELC/113999/2009), Threads (PTDC/EEAELC/117329/2010) and project PEst-OE/EEI/LA0021/2013.

978-1-4799-5313-4/14/$31.00 ©2014 IEEE

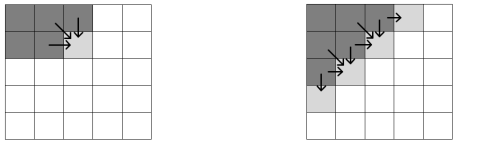
that are conventionally exploited by dedicated architectures of these particular domains, by considering a combination of two widely studied parallelization approaches: DLP tightly combined with Instruction Level Parallelism (ILP). While the former enables the computation of multiple data elements in parallel and it is commonly used in vector architectures, the latter implies the existence of multiple functional units concurrently computing different instructions, and it is frequently used in superscalar and in Very Long Instruction Word (VLIW) architectures. In particular, the ILP methods that are exploited in the proposed architecture are similar to those used in VLIW architectures, consisting in a single instruction issue with static scheduling. This permits a better usage of the functional units by concurrently assigning different instructions to different units, thus reducing the number and size of those units. The memory accesses delay is also highly mitigated by the inclusion of a Data Stream Unit that will enable a parallel memory access during the processing of the target algorithm, allowing an optimum exploitation of efficient DLP schemes for each algorithm. Hence, by taking advantage of both types of parallelism, the proposed architecture results in an innovative Vector/VLIW hybrid architecture, that is able of simultaneously computing a bundle of different instructions, each processing a different vector of data elements, while reducing and making a better usage of the available hardware, thus achieving a better energetic efficiency.

The proposed programmable hybrid Vector/VLIW architecture was prototyped on a Xilinx Zynq 7020 System on Chip (SoC). To evaluate the architecture, the experimental procedure was focused on sequence alignment algorithms, with a particular emphasis to the Smith-Waterman algorithm. The considered studies cover both performance and energy metrics for the proposed architecture and compare them to a dedicated ASIP specialized for this type of algorithm: the BioBlaze [6]; and an embedded processor frequently applied in those low-power application domains: the ARM Cortex-A9. According to the presented evaluation, the proposed architecture achieves a better performance-energy efficiency in comparison to the ARM processor and the BioBlaze, while operating at a much lower frequency.

II. Exploiting the parallelism of DP Algorithms

When solving a problem using a DP-based approach, it is first necessary to decompose it into a set of smaller sub-problems. This translates into computing the value of each cell in a n-dimensional matrix by relying on the value of pre-computed adjacent cells. In a 2D matrix, this typically results in horizontal, vertical and diagonal data dependencies from the top, left and top-left cells, respectively (see figure 1(a)). To maximize the processing efficiency and to minimize the number of dependencies, cell computations should be performed in parallel along the anti-diagonal (see Fig. 1(b)).

Exploiting this DLP along the anti-diagonal brings two problems as previously mentioned: harder memory organization/access and larger hardware requirements. While the former can be solved by implementing specialized memory-access units to gather cell values in non-adjacent memory positions, the latter requires a different type of parallelism. In fact, vector-only solutions will always result in low FU usage. For example, consider that vector processing is used



(a) DP left, top and top-left dependencies.

(b) DP cell parallelism.

Fig. 1. Dynamic Programming dependencies.

to compute the value of N cells in parallel, which requires a total of M vector instructions. Assuming the inexistence of any dependency and that only one of these operations is a square root, an utilization of $1/M$ is expected for all N parallel square root FUs. Naturally, it is possible to reduce the number of FUs (hardware requirements) by serializing the operation on the different vector elements. However this solution trades performance for hardware requirements, hence it is not ideal. The alternative is to also explore ILP. Since the operation over the vector elements along the anti-diagonal is independent, different cells can compute different instructions simultaneously. This not only increases the potential for additional parallelism, but also reduces the hardware requirements. The use of ILP is also supported by the common steps in a DP algorithm. Usually these steps consist in dependency loads followed by cell computations and finalized with the results storing. Assigning these different steps of the algorithms to different cells in the matrix (along the anti-diagonal) validates the ILP (different instructions operating over different cells) while also maintaining data coherence, given the independence between the cells.

There are several ways to explore ILP alongside DLP. In the proposed architecture, static ILP is explored since it requires less hardware control, thus achieving better energy efficiency. This is achieved by issuing an instruction bundle that is composed of several different instructions, each operating over a vector of independent elements (DLP) in different execution units. This way, instead of using a single large vector computing the same instruction (as is typical in vector architectures), we have several smaller vectors, each effectively computing a different instruction. In a DP algorithm this translates into processing cells in different steps of the algorithm without any data races between them. In fact, the data races will not occur as long as two conditions are met: all cells currently being processed are independent (which is true along the anti-diagonal); and the cells that are being processed in advance will never be dependencies of the results of cells in previous processing steps. In order for this second condition to be met, the cells being processed at the most down-left section of the anti-diagonal, should be in advance regarding the cells at the top-right section of the anti-diagonal (see Fig. 2).

To efficiently support DP algorithms and to explore DLP and ILP, the proposed architecture has the following requisites:

- Independent execution units to compute each instruction of the instruction bundle.
- A Data Stream unit to access the memory concurrently to the execution units.

Each execution units operates a different vector of cells and, given the independence between the cells, has its own

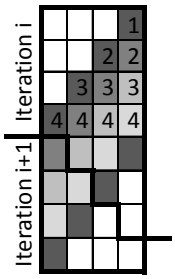


Fig. 2. Example of two iterations of a DP algorithm with 4 instructions per iteration and 4 elements being processed, with the processing along the anti-diagonal. Each number corresponds to one of the four instructions of an iteration, and each color corresponds to a different clock cycle.

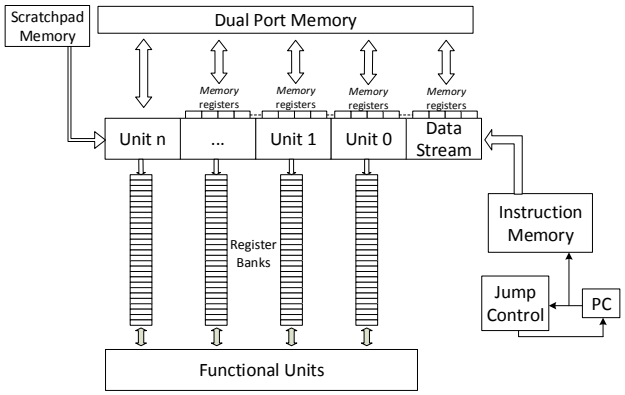


Fig. 3. Proposed hybrid architecture.

register bank. This locally stores the results that are generated each iteration of a DP algorithm, thus reducing memory access operations and improving the processing performance. However, since the bundle units that are in advance regarding the computation steps of the algorithms will generate dependencies that are required by other delayed units, a sniffing mechanism for a small subset of the register banks is necessary, in order to maintain the independence of the register banks while keeping data coherence and avoiding unnecessary memory accesses.

Although the register banks are used to store dependencies, it is still often required for DP algorithms to store some of their dependencies in memory, especially when they are required in a much later iteration of the algorithm. Therefore, the Data Stream unit is required to perform the necessary memory loads and stores in parallel to the execution units, minimizing the impact of these memory accesses on the processing performance. The performance peak is reached when the algorithm does not have any other unit accessing the memory at the same time as the data stream unit. Contrary to common VLIW architectures, here, all the existing units (executing units and the data stream unit) can access the memory, which requires a priority access list to avoid conflicts. Since the data stream unit main function is memory access operations, it has the top priority over all other units.

To further ease the memory access delay problem, a local fast (scratchpad) memory is also included, which is used to store constant values which required in several DP algorithms. These constant values are pre-fetched at the beginning of the computation and can only be accessed by the execution units (with a similar access priority list between them). Since it is a different memory, it can be accessed in parallel to the main memory, enabling both the Data Stream Unit and one of the execution units to perform a load or store instruction.

With the requirements listed above, the hybrid architecture can also be easily scalable in two distinct ways: by increasing the length of each execution unit and thus increasing the vector length (DLP); and by increasing the number of execution units, and thus increasing the number of parallel instructions (ILP). The first solution would mainly require an increase of the size of the functional units, while the second solution would require an increase in the number of functional units.

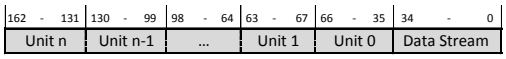
III. VLIW Architecture

Following the previous specifications, the proposed architecture (Fig. 3) is composed by a bundle of n execution units, each with an independent register bank, and a Data Stream Unit that communicates with the memory concurrently with the other units. It presents 4 typical pipeline stages: a `FETCH` stage where the next instruction is loaded from the instructions memory; a `DECODE` stage, where the fetched instructions are decoded; an `EXECUTE` stage where the FUs and memory operate the instructions; and a `WRITE-BACK` stage, where the results are written to the register banks. The pipeline also includes data forwarding mechanisms to minimize the number of stalls on the processor.
The architecture is also composed by *scalar* and *vector* functional units, by a Random Access Memory (RAM) and by a local fast memory. All these blocks can be accessed by any one of the n execution units, admitting that they are free to be accessed, i.e., no other unit is using them. The Data Stream Unit can only communicate with the RAM.

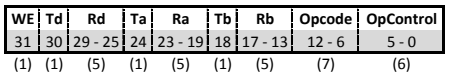
A. Instruction Set Architecture

Each execution unit instruction has a total width of 32 bits and the Data Stream Unit instruction has a width of 35 bits. The instruction bundle is divided into several different units as seen in Fig. 4(a). As previously mentioned, more execution units can be easily added to the architecture by widening the instruction bundle and adding register banks to the new units. Furthermore, these execution units can also be expanded to accommodate more words, by increasing their vector width. This would also require some modifications in the FUs and in the memory accesses in order to maintain compatibility. The former scalability solution is better suited to algorithms that require many instructions per iteration while the latter has a better use for algorithms that require less instructions and work with high volumes of data.

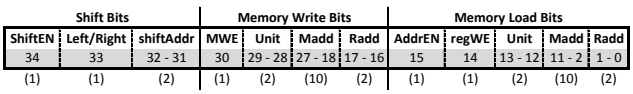
The execution units (Fig. 4(b)) are composed by the common register address fields (`Ra`, `Rb` and `Rd`), by a `WE` field that indicates when a register write is required, and by the instruction fields, namely the `opcode` field that selects between the different types of instructions (arithmetic/logical, control/branch and memory access) and the `Opcontrol` field, that identifies a certain modifier to the instructions (use of immediate or unsigned values in arithmetic/logical operations



(a) Instruction Bundle.



(b) Execution unit instruction.



(c) Data Stream unit instruction.

Fig. 4. Instruction words for the bundle and the composing units.

and the use of inequality comparisons for control operations). The `Opcontrol` field can also be used to distinguish more specialized instructions. Three more special control fields are also present: `Td`, `Ta` and `Tb`. The `Td` field enables a broadcast write (enabling a 3-way register write, relevant for DP algorithms that have up to 3 dependencies), while bits `Ta` and `Tb` are used to specify which part of the data is to be loaded or written to registers, for memory instructions that operate with dividable parts of data. A summarized version of the instruction set can be seen in Table I (the immediate and carry instructions are not depicted).

TABLE I. IMPLEMENTED REDUCED INSTRUCTION SET ARCHITECTURE. FURTHER INSTRUCTIONS CAN BE EASILY ADDED.

INSTRUCTION	**MNEMONIC**
Arithmetic and Logic Instructions	
Add, Subtraction	**SUM, SUB**
Maximum, Maximum and Move	**MAX, MAXMOV**
Multiplication	**MUL**
Comparison	**CMP**
Arithmetic and Logic Right and Left Shift	**SRA, SRL, SLA, SLL**
Logic OR, AND, XOR	**OR, AND, XOR**
Control Instructions	
Load Byte, Half-word, Data	**LB, LH, LD**
Index Memory address	**INDEX MADDR**
Index local memory address	**INDEX SADDR**
Local Memory Load	**SPAD LD**
Store Byte, Half-word, Data	**SB, SH, SD**
Control Instructions	
Delayed Branch	**BRD**
Delayed Branch Equal, Not Equal	**BEQD, BNED**
Delayed Branch Less Than, Greater Than	**BLTD, BGTD**

The Data Stream Unit has a different instruction format than the execution units and is depicted in Fig. 4(c). This instruction itself also explores ILP, since it computes 3 distinct and parallel operations: a memory load/register write (bits 15 - 0); a memory write (bits 30 - 16); and a register shift (bits 34 - 31). The memory access operations have priority over the memory access instructions in the execution bundles, as previously stated.

The load operation of the Data Stream Unit has an `AddrEN` bit to indicate if the Data Stream is accessing the memory address for a load operation. This prevents the Data Stream Unit to take over a load instruction when it is only operating the memory index. Both the load and write operations have an `Unit` field that allows selecting the execution unit's register banks.

The register shift operation is responsible for creating a register window mechanism on a smaller subset of registers in every execution unit data bank. This mechanism is depicted in Fig. 5 and can reduce the impact of memory accesses, by pre-loading a data value required in future iterations of the computation or by pre-storing a value to be later used in future iterations. The registers to be shifted are chosen by the `ShiftAddr` bits, from one of the periphery execution units (unit 0 or unit n) to the opposite unit, with the direction being chosen by the `Left/Right` field.

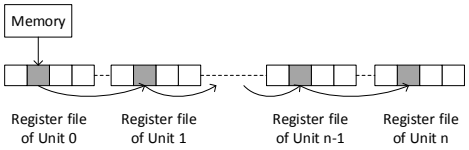


Fig. 5. Register Window.

B. Register Banks

Each of the previously mentioned execution units has its own private register bank of 28 registers and a separate small set of 4 *memory* registers, achieving a total of 32 registers.

The array of 4 *memory* registers can be read by all execution units, however, writing is only allowed for the shared register owner. These registers are also used by the Data Stream Unit to communicate with the memory (hence the *memory* name tag). Furthermore, sharing these registers between different execution units can also reduce the number of memory accesses, in situations where a dependency value loaded by one execution unit is required by other units. In these situations, the dependency values will only be loaded once, thus reducing the number of memory accesses. The Data stream unit has write priority over the execution units for these *memory* registers. This is done in order to prioritize the parallel memory access, reserving the remaining registers to store the intermediary results of the computations.

C. Functional Units

All register banks in all execution units share the FUs of the architecture. Since the number of available FUs is limited, in order to maximize processing performance, the execution units should not issue more operations of a given FU type than those available on the architecture. However, in case such instruction bundle is generated by the compiler, a structural conflict appears, resulting in the original instruction bundle being executed in multiple cycles (the conflicted operations are held and delayed until they can be executed).

The FUs are also prepared to operate with different numbers of vector elements. If the vector is composed by only one large element, the FUs behave like scalar units, each operating over single data elements. If the vector is composed by several smaller elements, the FUs will then behave like vector units, each operating over multiple data elements. The FUs implemented in the architecture consist of: Sum, Maximum, Shift, Logic and Comparison units.

D. Memory

In addition to the instructions memory, there are two distinct memories in the architecture: a dual port RAM memory with a write-only and load-only ports, and a local fast memory. The former can be accessed by the data stream and the execution units to store and read values, while the latter is a load-only memory that can only be accessed by the execution units. In fact, the Data Stream Unit is the only element of the architecture that can write into this local memory, storing constant values required by DP algorithms. These stores are done by memory mapping a section of the main RAM memory. The existence of two memories minimizes the delay introduced by concurrent memory accesses and also promotes a better data organization, by separating the constant data values of the algorithms from the constant changing intermediary results.

The memories have an access latency of 2 clock cycles: one cycle to index the correct address, and another cycle to load the value in the previous indexed address. In fact, there are two instructions to compute both steps of a memory load, enabling a parallel usage of a memory load instruction between two units: one indexing an address, and the other effectively loading a data word. This allows achieving a throughput of 1 clock cycle with a latency of 2 clock cycles, when loading a value from memory.

IV. Case Study: Smith-Waterman

In order to evaluate the architecture performance, the Smith-Waterman algorithm was considered as a good candidate, since it is widely used in bioinformatics for local sequence alignment of DNA, RNA or protein chains. In particular, we address the case study of embedded systems for DNA sequence alignment, such as for the development of biomarker detection SoCs. The SW algorithm is one of the most sensitive sequence alignment algorithms, but is also one of the slowest, operating in sequence banks that grow larger at a very fast rate [8]. The algorithm computes the optimal local alignment (zone of most similarity) between two DNA or protein sequences, with the help of a substitution score matrix and, originally, a general gap penalty function [2]. Gotoh [9] later improved the algorithm by using an affine gap penalty model, allowing multiple sized gap penalties. These improvements were used in the implemented algorithm.

Given a query sequence (Q) and a reference sequence (D) of size m and n respectively, a substitution score matrix (S_m) and a gap initialization and extension penalties of α and β respectively, the score matrix (H) can be computed by the following recursive relations:

$$H_{i,j} = max \begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i-1,j-1} + Sm(q_i, d_j) \end{cases} \quad (1)$$
$$H(i,0) = H(0,j) = 0$$

$$E_{i,j} = max \begin{cases} E_{i,j-1} + \beta \\ H_{i,j-1} + \alpha \end{cases} \quad (2)$$
$$E(i,0) = E(0,j) = 0$$

$$F_{i,j} = max \begin{cases} F_{i-1,j} + \beta \\ H_{i-1,j} + \alpha \end{cases} \quad (3)$$
$$F(i,0) = F(0,j) = 0$$

where the terms $E_{i,j}$ and $F_{i,j}$ correspond to the scores ending with a gap in the reference sequence and to the scores ending with a gap in the query sequence, respectively. These terms correspond to the vertical ($E_{i,j}$) and to the horizontal ($F_{i,j}$) dependencies of the DP algorithms, that were previously mentioned in the second section. $H_{i,j}$ (1) represents the local alignment score involving the first i symbols of Q and the first j symbols of D. After the score matrix is filled, a traceback runs over the matrix returning the local alignment. This score takes into account both the vertical and horizontal dependencies and also the diagonal dependency, given by the equation $H_{i-1,j-1} + Sm(q_i, d_j)$ in (1). These three dependencies lead to a parallelism along the anti-diagonal previously mentioned.

A. Implementation

Following the previous Smith-Waterman equations, the proposed architecture can simulate one iteration of the algorithm by computing the sequence of operations in table II.

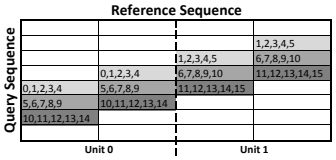
TABLE II. Smith-Waterman algorithm operations

Instruction	Description
Comparison	Compares the query and reference symbols and returns the substitution score matrix index
Score Load	Loads the substitution score
Recursion Sums	3 sums/subtractions that compose the main recursion of the algorithm
Maximum 1	Maximum operation between two of the previous results
Maximum 2	Final maximum operation between the previous result and the remaining recursion operation result

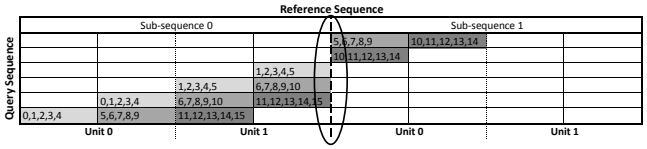
To perform the computation, each execution unit gets assigned with a fixed number of cells during each iteration of the algorithm. The number of cells depends on the width of the sequence symbols and scores used, i.e., the vector element's size. Although the processor is configurable to admit other setups, the described implementation uses 32-bit vectors, each composed of 4 8-bit words.

The alignment procedure is performed along anti-diagonal (see Fig. 6(a)). To explore ILP, the execution units that are more advanced in the query sequence are computing advanced instructions of the algorithm iteration. This requires new query symbols to be loaded every iteration for the execution unit in advance, and a query sequence shift for the remaining units to allow the remaining execution units to reuse the symbols that are pre-loaded by the unit in advance. The Data Stream Unit is used to pre-load the next symbols and to shift the symbols along the units, in parallel to the algorithm computations. This register window mechanism is the same as the one in Fig. 5.

The reference sequence, also stored in memory due to its size, will require to be loaded accordingly to the necessity of an execution unit. Given the processing along the query sequence, this will only occur when one unit reaches the end of the query sequence, requiring new reference symbols to continue its computations. Since the Data Stream unit will be occupied with a different load operation, instead of a register window



(a) Instruction Delay between execution units.



(b) Critical section between two sub-sequences of the reference sequence.

Fig. 6. Instruction Delay and Critical Section for an example case where each unit computes 2 cells (i.e., uses a vector of two elements). Both tables represent the computation of the DP matrix shown in Fig. 1(a), where each cell was annotated with the clock cycles to execute the instructions that compute the cell value.

scheme, an additional load instruction on the execution unit that reached the end of the sequence is done. This will introduce additional cycles to the overall processing. However, the performance impact caused by these load instructions is very small, considering the large size of both sequences.

Mapping the operations in table II and the considerations taken above to the instruction set architecture, results, during the stationary phase, in an average of 5 instructions for an execution unit to complete an iteration of the algorithm. The pseudo-code can be seen in Fig. 7 for the linear model of the algorithm (the affine model has additional gap control computations running in parallel to the same instructions of the linear model), where both the instructions for the Data Stream unit and the executing units can be seen.

	Pseudo-Code	Proposed Architecture
Stationary phase	**Execution Units**	**Execution Units**
	Sm = Load(Q,R)	
	sumA = rD + Sm	**INDEX SADDR** R29,R30
	sumB = rH - gap	**SPAD LD** R1,Spad(IMR)
	sumC = rV - gap	**SUM** R(0,8,12),R(1,9,13),R(2,10,14)
	sumA = MAX(sumA, sumB)	**MAXMOV** R0,R0,R8, mov(R2, R10)
	rD = rD2	**MAX** R(10,14,),R0,R12
	max = MAX(sumA, sumC)	
	rD2 = sniff & max	
	if end of query sequence	**if end of query sequence**
	R = Load(R+1)	**INDEX MADDR** #Ra,#Rb
		LD R30,M(IMR)
	Data Stream Unit	**Data Stream Unit**
	rHD2_0 = Load (rH_3)	
	Q = Load(Q+1)	**INDEX MADDR** index, (#H_D)
	Store (max_3)	**LD | INDEX MADDR** R7, M(index) | index, (#Q)
	Store (gap_3)	**LD** R28, M(index)
	gap_0 = Load (gap_3)	**ST | INDEX MADDR** M(H_D3), u3_R14 | index, (#gap)
	Shift (Q)	**ST | LD | SHIFT** M(gap3), u3_Rgap | Rgap, M(index) | shift_r(29)

Fig. 7. Pseudo-code definition for the Data Stream and execution units for the linear model of the algorithm.

It is also important to notice that, to solve cell dependencies, some special mechanisms are included. The vertical dependency of the cells is computed in the same execution unit, hence will not require any additional mechanisms. However, both diagonal and horizontal dependencies of the cells on the left boundary of each execution unit (n), require the value of the cells previously computed by unit $n-1$. To reduce memory bandwidth requirements, a sniffed mechanism was implemented that allows an execution unit to sniff the values written to the register file by an adjacent execution unit. These dependencies, when occurring inside the same execution unit, will not require any additional mechanisms , since the unit has access to all cell values required for the computations. There is however a special case for the execution unit 0. It requires the horizontal and diagonal dependencies to be loaded from memory, since these dependencies were computed in a previous distante iteration (see Fig. 6(b)). Therefore, the Data Stream unit will always write the right boundary cell value of the last execution unit n to memory, and load the respective cell value to unit 0 when required. The diagonal dependency, unlike the other two dependencies, will require two registers (instead of one) to be stored. The first register stores the diagonal dependency to be used during the current iteration while the second stores the diagonal dependency to be used in the iteration after it. This is due to the fact that the computed cell value is only used as a diagonal dependency two iterations after it has been computed.

The affine gap model will also require a mechanism similar to the horizontal and vertical dependencies. Since this model takes into account two distinct gap values, an initialization value and an extension value, all execution units will have two registers in their register bank with both gap values constantly stored. During the maximum operations, an auxiliary register will store the information regarding which dependency originated the max result. If it is a vertical or horizontal dependency, the auxiliary register will compare its previously stored value, and check if the new result is a gap extend or initialization, updating its value accordingly. This way, during the recursion operations in the following iteration, the correct gap value to be used is already stored in the register bank. For execution unit 0, since the auxiliary register that indicates the type of gap for the horizontal dependencies lies in the register bank from unit n, they need to be stored in memory after their computation, and loaded when required by unit 0. These stores and loads are done by the Data Stream unit during the algorithm iterations. The remaining bundle units will sniff the auxiliary registers for the horizontal dependencies gap types, in the same way they do for the horizontal dependencies.

B. Experimental Results

To evaluate the proposed architecture a performance analysis as well as a power analysis, are presented in this subsection. As previously referred, our case study targets low-power embedded systems for biomark detection SoC. Thus, we evaluate the implementation of the proposed architecture on a Zynq SoC 7020 composed by a dual-core ARM Cortex-A9 running at 533 MHz and programmable logic. Therefore, to evaluate the proposed architecture, we compare its performance and energy-efficiency with that of the embedded ARM Cortex-A9 processor. To make the comparison fair, we use the state-of-art implementation of the Smith-Waterman algorithm [4], which uses SIMD ISA extensions to exploit the parallelism of the SW algorithm. The ARM Cortex-A9 cores support out-of-order execution, with dual instruction issue and 128-bit SIMD extensions. This allows issuing up to 2 instructions per clock cycle, each processing up to 16 cells in parallel. The proposed architecture is also compared to the BioBlaze [6], a dedicated ASIP architecture running at 158 MHz, which uses a 128-bit SIMD modified extension, with each instruction processing 16 cells in parallel. The BioBlaze [6] was also prototyped in a

Zynq SoC in order to maintain a fair comparison. We compare one core of both these two architectures with one core of the proposed processor, issuing one bundle of instructions to 4 execution units, each using vectorial instructions to process 4 cells in parallel. This allows to compute 16 cells in parallel. The proposed architecture and the BioBlaze [6] were both synthesized and mapped to the programmable logic by using the Xilinx ISE 14.5.

The proposed architecture consists of one Data Stream unit and 4 execution units with a vector width of 32 bits, where each vector is composed by 4 8-bit elements. This achieves 16 cells computations in parallel, 4 in each execution unit. The symbol, score and gap values were all set to 8 bits as well and, since both memories have data widths of 32 bits (the same width as the vectors), four values can be written or loaded from memory with a single write or load instruction. This also applies to the registers in the register banks, where each registers stores 32 bits of data, or up to 4 different 8-bit values. As previously mentioned, if a higher resolution was required, the vector elements size (and the symbol, score and gap values) could be easily increased to 16 or 32 bits, reducing the number of computed cells per bundle unit to 2 and 1 respectively.

As can be observed in table III, the proposed architecture uses 6% of the Slice Registers, 50% of the Slice LUTs, and 5% of the BRAMs, achieving a maximum operating frequency of 98.5 MHz.

TABLE III. HARDWARE RESOURCES, OPERATION FREQUENCY AND POWER ESTIMATION OF THE PROPOSED ARCHITECTURE

Hardware Resources	Used	Total	Utilization
Slice Registers	7135	106400	6%
Slice LUTs	26725	53200	50%
36-bit Block RAMs	7	140	5%
Frequency	98.5 MHz		
Power	0.584 W		

Using the Xilinx Power Estimation tool, we further estimate the power consumption of the proposed processor (see table III). Assuming worst-case conditions for flip-flop and memory updates, it results in a power consumption of 0.584 W.

A scalability test was also performed, by changing the size of the vector width from 32 to 40 bits or by including an additional execution unit. Both cases increase the raw throughput by 25%, while the hardware resources scales almost linearly. The increase of the vector width results in a 21.4% and 24.6% increase of slice registers and LUTs, respectively, while the addition of one execution unit results in an increase of 23.3% and 29.9% in slice registers and LUTs. The number of Block RAMs is only affected by the changes to the vector width, increasing by one unit for every 16 bits added to the length of the vector. The operating frequency drops to 66.4MHz (32.6%) when the vector width increases to 40 bits, and to 74.0MHz (26%) with the addition of an execution unit.

To further evaluate the architecture we compare its performance against the ARM Cortex-A9 and the BioBlaze [6]. For this, a DNA dataset composed of several sequences, ranging from 128 to 16384 elements, and a set of query sequences of length ranging from 20 to 2276 elements was used. The reference sequences correspond to twenty indexed regions of the Homo sapiens breast cancer susceptibility gene 1 (`BRCA1`gene) (`NC_000017.11`). The query sequences were obtained from a set of 22 biomarkers for diagnosing breast cancer (`DI183511.1` to `DI183532.1`) and a fragment, with 68 base pairs, of the `BRCA1` gene with a mutation related to the presence of a Serous Papillary Adenocarcinoma (`S78558.1`).

1) Performance Evaluation: For the proposed architecture, the local fast memory values were pre-loaded before the initialization of the algorithm, and only the algorithm steps were accounted for the performance analysis.

Accurate clock cycle measurements of the required time to execute each biological sequences analysis in the proposed platform were achieved by using the Xilinx ISim. On the Bioblaze [6], the clock cycle measurements were achieved by using Modelsim SE 10.0b. For the ARM Cortex-A9, the system timing functions were used to determine the total execution time of the DNA sequence alignment. To improve the measurement accuracy, several repetitions of the same alignment were done. The obtained values were subsequently divided by the number of repetitions and the processor clock frequency.

To compare the proposed architecture with the ARM Cortex-A9 and the BioBlaze [6], several metrics are used, namely: the processor performance measured in both number of Clock Cycles per Cell Update (CCPCU) and number of Cell Updates Per Second (CUPS); the energy efficiency measured in Cell Updates per Joule (CUPJ); and the equivalent Energy-Delay Product (EDP) metric, measured in Cell Updates per Joule-Second (CUPJS).

The average execution time (in number of clock cycles) to execute the DNA sequence alignment is presented in table IV for both the proposed architecture, the ARM Cortex-A9 and the BioBlaze [6]. The resulted speedups can be observed in the respective columns, which accounts for the affine model of the algorithm. A better efficiency comparison metric is presented in table V, where the clock cycles per cell update can be seen. These values were obtained by dividing the total number of clock cycles (c) by the length of the reference and query sequence (m and n respectively) - $c/(m \times n)$. It can be seen that the proposed architecture achieves a speedup of 13.7x against the ARM Cortex-A9, even though the latter processor can issue two instructions per clock cycle and operate on 128-bit words, i.e., 16 cells per vector. Against the Bioblaze [6], a speedup of 5.44x is achieved. This shows the advantages of the proposed architecture, regarding the case where only data-level parallelism is explored.

2) Performance and Energy Efficiency Evaluation: In addition the the presented evaluation comparisons, several efficiency metrics are also used to study the computational and energy efficiency of the proposed architecture: *i)* the attained raw throughput, *ii)* the energy efficiency, and *iii)* the performance-energy efficiency.

To compare the attained raw throughput, the Cell Updates Per Second (CUPS) metric is presented. This metric accounts the total number of cells (given by the length of the query sequence (m) times the length of the reference sequence (n)) that are updated in a corresponding runtime (t), in seconds. Therefore, the CUPS metric is obtained as $(m \times n)/t$ and can be seen in table V. This metric accounts for the maximum

TABLE IV. EXECUTION TIME (IN CLOCK CYCLES) FOR DIFFERENT DNA QUERY SEQUENCES MATCHED AGAINST A 4092 ELEMENT REFERENCE SEQUENCE, FOR THE CONSIDERED EXECUTION PLATFORMS

Clock Cycles [$\times 10^6$]					
Query Size	**ARM CortexA9 (NEON)**	**Speedup**	**BioBlaze [6]**	**Speedup**	**Proposed Architecture**
20	1.154	32.971	0.307	8.771	0.035
68	1.373	15.256	0.555	6.167	0.090
74	1.339	13.804	0.543	5.598	0.097
85	1.470	13.243	0.631	5.685	0.111
94	1.373	11.254	0.627	5.139	0.122
685	6.303	7.195	3.375	3.853	0.876
1861	16.262	6.833	8.848	3.718	2.380
2276	19.491	6.696	10.744	3.691	2.911

TABLE V. PERFORMANCE AND ENERGY EVALUATION RESULTS

Evaluation metrics	**ARM Cortex-A9**	**Bio Blaze [6]**	**Proposed architecture**
Clock Cycles per Cell Update (CCPCU)	4.29	1.70	0.31
Mega Cell Updates per Second (MCUPS)	124.24	62.94	315.18
Total Power Consumption [W] (TPC)	1.00	0.30	0.58
Mega Cell Updates per Joule (MCUPJ)	130.78	331.93	539.69
Peta Cell Updates per Joule.Second (PCUPJS)	127.47	175.64	412.43

operating frequency of each implementation platform, namely 533 MHz for the ARM Cortex-A9, 158 MHz for the BioBlaze [6] and 98.5 MHz for the proposed architecture.

The energy efficiency study was performed by using the Xilinx Power Estimator tool to obtain the power values for the Zynq SoC. The energy efficiency metric adopted, Cell Updates per Joule (CUPJ) is given by the total number of processed cells, divided by the total consumed energy (TPC metric in table V), and can be seen in table V. The adopted performance-energy efficiency metric is given in Cell Updates per Joule-Second (CUPJS) and can be regarded as an inversion and normalization of the commonly used Energy-Delay Product (EDP) metric. In fact, while the EDP is generally given by the product of the total energy consumption and the corresponding runtime, the adopted CUPJS is obtained by inverting the EDP and by multiplying it with the total number of processed cells. The results obtained for this metric can also be seen in table V.

Analyzing the MCUPS metric in table V, it can be seen that the proposed architecture achieves a throughput superior to that of both the ARM Cortex-A9 and the Bioblaze [6], even with a lower frequency than both architectures. The total power consumption of the proposed architecture is also lower than the power consumption of both the ARM Cortex-A9 and the BioBlaze [6], leading to an energy efficiency of 4.13x and 1.63x greater than the ARM and the BioBlaze [6], respectively, and a performance-energy of 3.24x and 2.35x greater than the ARM and the BioBlaze [6], respectively, as can be seen in the correspondent rows (MCUPJ and PCUPJS respectively) in table V.

From the presented results, it is possible to conclude that the proposed architecture complies with all the requisites to be embedded in low-power autonomous platforms, while providing high performance and support to a wide range of DP algorithms, which capabilities are often only offered by state-of-art GPPs.

V. CONCLUSION

The complexity of DP algorithms often lead to long execution times, which results in large energy consumptions. Exploring DLP in state-of-art architectures has proven critical to maximize the performance, which is achieved by computing several data elements in parallel. Given the number of dependencies found in DP based algorithms, vector processing should be made by processing the cells along anti-diagonal. However, this requires non-coalesced memory accesses, which drastically decreases the performance in GPPs implementations, leading to non-optimal and not fully parallel solutions. By exploiting DLP and ILP, the proposed architecture proves that it is possible to reduce the hardware resources requirements, while improving the performance and reducing the energy consumption.

To quantitatively evaluate the proposed architecture, we have compared it against a general purpose ARM Cortex-A9 and a dedicated ASIP (BioBlaze [6]). The evaluation tests performed in all architectures show a clearly advantage of the proposed architecture in both performance and energy efficiency. In fact, a speedup of 2.54 and 5.01 was obtained against the ARM and the BioBlaze [6], respectively, using a raw throughput metric, and the performance-energy was 3.24x and 2.35x better than the ARM and the BioBlaze [6], respectively.

The results obtained show that the proposed architecture has all the traits to be embedded in low-power autonomous platforms while providing high performance, which is often only seen in state-of-art GPPs. Furthermore, the architecture is easily scalable, by adding new execution units, or by widening the existent ones, enabling an increase in performance with a relatively lower impact on the additional hardware resources and thus the energy efficiency.

REFERENCES

[1] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[3] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *Information Theory, IEEE Transactions on*, vol. 13, no. 2, pp. 260–269, 1967.

[4] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[5] S. R. Eddy, "Profile hidden markov models." *Bioinformatics*, vol. 14, no. 9, pp. 755–763, 1998.

[6] N. Neves, N. Sebastiao, A. Patricio, D. Matos, P. Tomás, P. Flores, and N. Roma, "Bioblaze: Multi-core simd asip for dna sequence alignment," in *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*. IEEE, 2013, pp. 241–244.

[7] T. Rognes and E. Seeberg, "Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.

[8] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler, "Genbank," *Nucleic acids research*, vol. 28, no. 1, pp. 15–18, 2000.

[9] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of molecular biology*, vol. 162, no. 3, pp. 705–708, 1982.