Accelerating Phylogeny-Aware Short DNA Read Alignment with FPGAs

Nikolaos Alachiotis, Simon Berger, Alexandros Stamatakis
The Exelixis Lab, Scientific Computing Group
Heidelberg Institute for Theoretical Studies
Heidelberg, Germany
Emails: {Nikolaos.Alachiotis, Simon.Berger, Alexandros.Stamatakis}@h-its.org

***Abstract*—Recent advances in molecular sequencing technology have given rise to novel algorithms for simultaneously aligning short sequence reads to reference sequence alignments and corresponding evolutionary reference trees. We present a complete hardware/software implementation for the acceleration of a program called PaPaRa, a newly introduced dynamic programming algorithm for this purpose.**

We verify the correctness of the proposed architecture on a real FPGA and introduce a straight-forward communication protocol (using gigabit ethernet) for seamless integration with the encapsulating steering software that is executed on a PC processor. The hardware description *and* the software implementation are freely available for download.

When mapped to a Virtex 6 FPGA, our reconfigurable architecture can compute 133.4 billion cell updates per second for the novel, tree-based alignment kernel of PaPaRa. Compared to PaPaRa, running on a 3.2GHz Intel Core i5 CPU, we obtain speedups for the alignment kernel, that range between 170 and 471. For the entire application, that is, the alignment kernel and the trace-back step, we obtain speedups between 74 and 118.

***Keywords*-FPGA, dynamic programming, multiple alignment, phylogenetic inference**

I. INTRODUCTION

Significant advances in molecular wet-lab sequencing techniques, that is, methods for determining the order of nucleotides in a DNA molecule, have led to a tremendous biological data flood in recent years. The term `short read` refers to DNA sequence data that are produced by a new-generation sequencer. Current state-of-the-art pyrosequencing technologies can generate between 100,000 to 1,000,000 short reads. The read lengths typically vary between 30 and 450 nucleotides. To allow for an efficient and accurate phylogenetic analysis of such short read samples, novel maximum likelihood-based methods [1] have recently been introduced ([2], [3], [4]). These approaches, known as phylogenetic placement algorithms, assign the short reads to a fixed, given reference phylogeny, that is, an unrooted phylogenetic (evolutionary) tree which is based on a given multiple sequence reference alignment. Before applying one of the above phylogenetic placement algorithms, all short reads must be aligned to the reference alignment.

Here, we present a FPGA-based system for the acceleration of PaPaRa [2] (PArsimony-based Phylogeny-Aware short Read Alignment), a novel method for aligning short reads to a fixed reference alignment that also uses the information contained in the respective evolutionary reference tree. Although short reads can be aligned with respect to a reference alignment using the HMMALIGN tool (part of the HMMER [5] tool suite), PaPaRa outperforms HMMER in the context of phylogenetic short read placement, because the tree structure (phylogeny) is also incorporated by PaPaRa [2] for the alignment process. Like most alignment methods, PaPaRa is based on a dynamic-programming algorithm. The underlying principle is similar to the Smith-Waterman algorithm [6], with affine gap penalties [7]. However, the specific alignment kernel in PaPaRa is used to align a sequence (a short read) against an ancestral state vector that is derived from varying positions (branches) in the phylogenetic reference tree.To this end, compared to the Smith-Waterman algorithm, PaPaRa implements a unique alignment kernel and scoring scheme. Moreover, the PaPaRa alignment kernel is 'one-sided', that is, gaps can only be inserted into the query sequence and *not* into the reference alignment.

The source code of PaPaRa is available as part of RAxML [8] (http://wwwkramer.in.tum.de/exelixis/papara.tar.gz), a popular likelihood-based phylogenetic inference program. Profiling of the PaPaRa source code revealed that the scoring function (the alignment kernel) accounts for 98% to 99.5% of overall execution time. To date, reconfigurable devices have already successfully been deployed to accelerate Bioinformatics applications including phylogenetic inference kernels [9] and a plethora of pairwise sequence alignment algorithms [10]. Since PaPaRa uses phylogenetic inference and alignment kernels we explore FPGA technology to accelerate the code.

The FPGA-based system we introduce here is able to reduce the execution time of the PaPaRa kernel by up to 748 times compared to executing PaPaRa on an Intel Core i5 CPU running at 3.2GHz. The underlying system architecture of our design is to offload the scoring function (alignment kernel) calculations to dedicated hardware components (henceforth denoted as: Score Processing Units, SPUs) on the FPGA. The software implementation on the PC side is used to orchestrate computations, collect the scores, and perform the final trace-back step of the alignment algorithm. To allow for reproduction of all results in this paper the

software implementation and the hardware description of the FPGA design are available as open-source code at: http://wwwkramer.in.tum.de/exelixis/FPGA_papara.tar.gz.

The remainder of the paper is organized as follows: Section II describes the PaPaRa algorithm. In Section III, we review related work on FPGA-based accelerators for standard alignment algorithms. The reconfigurable architecture of the Score Processing Unit (SPU) is described in Section IV while Section V provides a detailed description of the verification process and the design of a fully functional FPGA-based system. In Section VI we provide performance results for our FPGA accelerator. We conclude in Section VII.

II. The PaPaRa Algorithm

The algorithm aligns short reads, henceforth denoted as query sequences (QS), against ancestral state vectors derived from the reference multiple sequence alignment (RA) and the corresponding phylogenetic reference tree (RT) that has been inferred using the RA. In a phylogenetic tree, known sequences (e.g., nucleotide or amino-acid sequence data) of living species (taxa) are assigned to the leaves (tips) of the tree. The internal nodes correspond to hypothetical common ancestors of the subtrees they define. Because the real sequences at the ancestral nodes are not known, different methods for representing the inherent uncertainty of ancestral states (ancestral sequences) have been introduced in the context of functions for scoring alternative phylogenetic trees (different evolutionary hypotheses).

The maximum parsimony (MP) method is a widely used optimality criterion for phylogenetic inference. Given a fixed multiple sequence alignment, the idea is to find the phylogenetic tree which explains the alignment by the least number of mutations. For nucleotide sequences (DNA/RNA) the alphabet contains four characters: A (Adenine), C (Cytosine), G (Gouanine), and T (Thymine). The computation and representation of ancestral state vectors under parsimony is described in [11]. The key point is that for DNA data, every edge (branch) b in the RT can be represented by a parsimony state vector $A_b = A_b^1, ..., A_b^n$, where each individual A_b^i represents the parsimony state of alignment site i (i.e., column i) of the RA. The individual entries in the parsimony state vector consist of bit vectors $A_b^i = a_b^i(A), a_b^i(C), a_b^i(G), a_b^i(T) \in [0,1]^4$, where the individual bits in a_b^i correspond to the four DNA characters. This is a key difference to the simpler case of pair-wise sequence alignment, where both input sequences can only have one character per site. Here, every site in A_b can be either an A, C, G, or T, or any possible combinations thereof (i.e., an ancestral site can be an A or a T), to reflect the inherent uncertainty of ancestral sequence states.

In PaPaRa all QS are aligned against all ancestral state vectors for all edges of the RT. The ancestral state vectors are obtained by successively (one at a time) inserting (and removing again) an additional node ('virtual root') into each edge of the RT. The ancestral state vector is then calculated for each virtual root (each edge of the RT) by combining the ancestral states of the left and right internal nodes that are attached to the branch under consideration. In addition to the ancestral state vectors, PaPaRa computes and uses an additional, dedicated signal which provides information about the distribution of gaps in the RT. This signal consists of a supplementary flag (CGAP) for each site i which is used to appropriately adapt the scoring function of the dynamic programming implementation. The CGAP signal is calculated along with the ancestral state vectors, according to the rules described in [2].

Figure 1. Example of a dyamic-programming matrix for aligning a QS against an ancestral state vector. The positions for which the QS characters match the bits in the ancestral state vector are highlighted.

The alignment scoring function is provided in equation 1. The equation recursively defines the score of the dynamic-programming matrix cell $D^{i,j}$ in column i and row j for aligning site A^i of the ancestral state vector against site B^j in the QS.

$$\begin{aligned}
CG^i &= \begin{cases} 10 & \text{if CGAP is set for site } i \\ 0 & \text{otherwise} \end{cases} \\
(GP_{OE}^i, GP_E^i) &= \begin{cases} (2,1) & \text{if } CG^i = 0 \\ (0,0) & \text{otherwise} \end{cases} \\
S^{i,j} &= \begin{cases} 0 & \text{if } A^i \text{and} B^j \text{match} \\ 3 & \text{otherwise} \end{cases} \\
D_E^{i,j} &= min \begin{cases} D^{i-1,j} + GP_{OE}^i \\ D_E^{i-1,j} + GP_E^i \end{cases} \\
D^{i,j} &= min \begin{cases} D^{i-1,j-1} + S^{i,j} + CG^i \\ D_E^{i,j} \end{cases} \qquad (1)
\end{aligned}$$

For instance, an ancestral state A^i at site i with $A^i = [1100]$ means that, As and Cs in the QS under consideration can be matched against alignment site i of the ancestral state without incurring a mismatch penalty. Thus, for scoring mismatches via the respective function $S^{i,j}$, the default mismatch penalty of 3 will be used, unless the bit corresponding to the character at position j in the QS is set in A^i. When this matching condition is met, the score returned by $S^{i,j}$ is

0. The special way in which match and mismatch penalties are treated along with the phylogeny-aware adaptive scoring scheme (CG^i) in PaPaRa represent a key difference to standard dynamic programming alignment methods. Furthermore, alignments are one-sided, which means that gaps can only be inserted into the QS. Figure 1 depicts an example of the dynamic programming matrix. The Figure shows the bit-vector representation of an ancestral state vector and the positions where the QS characters match the ancestral state. For the sake of simplicity, the additional CGAP signal is not shown in the example.

The PaPaRa algorithm consists of two phases. Initially, all QS are aligned against all ancestral state vectors in the RT. For each QS only the best alignment score is retained. Thus, given an RT with r taxa, m sites, and q QS, the program needs to execute $O(rq)$ alignment steps or $O(rqm^2)$ operations. During the initial phase, the actual alignments are not generated (i.e., the 'traceback' step is not performed) for performance reasons. During the second phase (given the best scores for all QS), the actual alignments are generated by aligning each QS again, but only against the respective ancestral state vector that produced the best score for the QS during the alignment step. As already mentioned, the initial (all QS against all ancestral probability vector) alignments account for more than 98% of overall runtime. Thus, this initial alignment phase represents the natural candidate for a FPGA-based acceleration.

III. Related Work

FPGAs have already successfully been used to accelerate DNA and protein alignment methods that rely on dynamic programming algorithms (e.g., the Smith-Waterman [6] algorithm). While there already exists a comprehensive bibliography dealing with this general and important topic, we are not aware of any recent papers that specifically focus on the acceleration of short read alignment against fixed reference alignments.

In [12], Li *et al.* presented a FPGA-based acceleration of the Smith-Waterman algorithm. They obtained a speedup of 160 compared to a C software implementation running on the same FPGA (Altera Stratix EP1S40 FPGA) on an Altera Nios II soft processor. The reconfigurable hardware part designed to accelerate dynamic programming matrix cell updates had a maximum operating clock frequency of 3.1 MHz and a peak performance of 24.5 million CUPS (dynamic programming cell updates per second).

Yu *et al.* [13] presented an architecture for the Smith-Waterman algorithm based on a systolic cell array. Due to the efficiency of their design, they managed to instantiate 4032 processing elements on a Xilinx XCV1000E-6 FPGA. Operating at a clock frequency of 202 MHz, their system achieved a performance of 814 billion CUPS.

Several alternative implementations for accelerating the Smith-Waterman algorithm using FPGAs ([14], [15]), vector operations on x86 CPUs [16], and GPUs (e.g., using CUDA [17]) exist. However, because the PaPaRa alignment kernel differs significantly from the standard Smith-Waterman implementation, we omit a more detailed review at this point. There also exists related work on accelerating the (more complex) dynamic programming kernel of HMMER [5]. For an overview of FPGA accelerator architectures for the Virterbi algorithm used in HMMER, please refer to [18]. Performance results vary between 0.7 and 20 million CUPS.

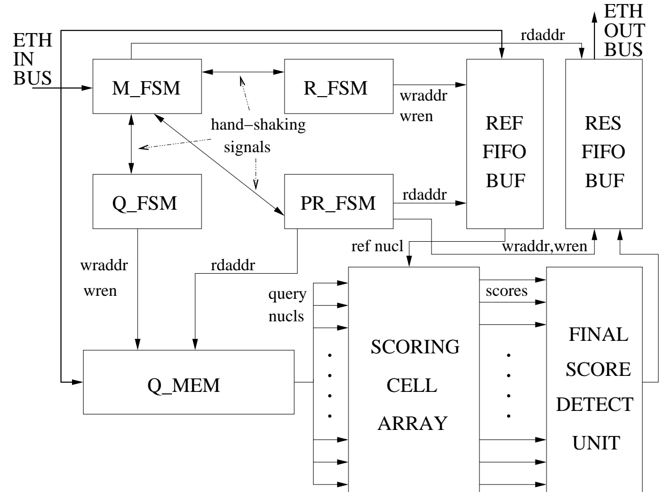

Figure 2. Top-level block diagram of the Scoring Processing Unit.

Note that, PaPaRa is currently the only dedicated algorithm for aligning QS against a multiple sequence alignment and a corresponding phylogenetic reference tree. Existing alternative algorithms that could be used in this context are based on aligning QS against a flat (non-phylogenetic) profile that is derived from the multiple sequence alignment without taking the phylogenetic tree into account. Existing tools which can be used for sequence/profile alignment are HMMAlign [5], MUSCLE [19], and MAFFT [20].

In addition to implementing this novel algorithm, our hardware implementation is also unique because we deploy the UDP-IP protocol over standard gigabit ethernet for PC-FPGA communication [21]. This allows for a seamless integration of the FPGA accelerator for the alignment kernel with the PaPaRa algorithm running on a standard Linux PC. Related FPGA implementations of the Smith Waterman algorithm used FPGA boards that were connected to the PC via the PCI bus [22] or directly via the CPU bus [23].

IV. The SPU Architecture

In the following we present the reconfigurable architecture of the Score Processing Unit (SPU). Figure 2 depicts the block diagram of the top-level design.

The entire control of the SPU is orchestrated by 4 finite state machines (FSMs) that are located in the top left corner

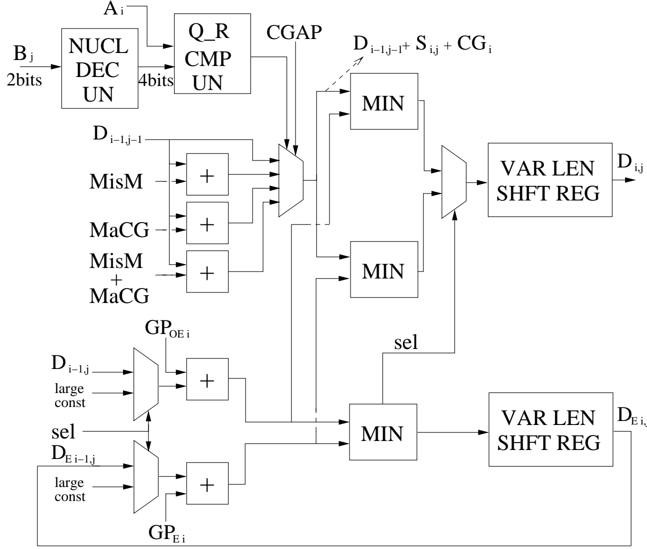

Figure 3. The scoring cell (*SC*) architecture.

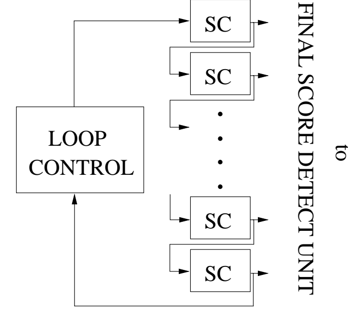

Figure 4. Loop control mechanism for accommodating query sequence lengths that are larger than the number of available *SCs*.

of Figure 2. The FSMs operate in a master-worker scheme; the master FSM (*M_FSM*) initiates and synchronizes the operation of the three worker FSMs (*Q_FSM*, *R_FSM* and, *PR_FSM*).

The *Q_FSM* is triggered when a QS is received. The main purpose of the *Q_FSM* is to generate the write-enable signal as well as the correct write addresses for the *Q_MEM* memory with a capacity of 200x16 bits. In each of the 16 memory lines, a total of 100 QS nucleotides are stored. The *Q_MEM* memory is used to store a single, incoming QS until an ancestral reference state vector (RS) arrives against which the QS shall be aligned. When the QS is stored in memory, the *Q_FSM* sends a signals to the *M_FSM* which then switches to reference-awaiting state.

Similar to the *Q_FSM*, the *R_FSM* is responsible for storing the RS in the RS FIFO buffer (*REF_FIFO_BUF*). The main difference between the *R_FSM* and the *Q_FSM* is that, once, the first position of the RS FIFO buffer has been filled, that is, the first state (A^0) for the first site of the RS has arrived, the *R_FSM* immediately notifies the master FSM.

The *M_FSM* interprets the notification from the *R_FSM* as a signal to start the actual computations and therefore triggers the processing FSM (*PR_FSM*). The *PR_FSM* generates read addresses for the *Q_MEM* and *REF_FIFO_BUF* memories as well as all the required signals for the operation of the *SCORING_CELL_ARRAY* which represents the computational kernel of the *SPU*.

The *SCORING_CELL_ARRAY* consists of 100 scoring cells (*SCs*) that operate in parallel. Figure 3 illustrates the architecture of the *SC*. The output ports of each *SC* are connected to a neighboring *SC* as well as to the *FINAL_SCORE_DET_UN* unit (see Figure 2) which selects the output of the *SC* that corresponds to the last line of the matrix and also determines the minimum value in that last line. When the entire matrix has been calculated, the *FINAL_SCORE_DET_UN* unit then writes the minimum value to the *RES_FIFO_BUF* output fifo buffer.

To accommodate scoring requests for query sequences whose length exceeds the fixed number of *SCs* available in the *SPU*, we also integrated a loop control module. Figure 4 provides an abstract representation of this subsystem. Based on the current position (index) of the nucleotides in the query sequence that are provided as input at each clock cycle, the *Loop_Control* module either feeds the output of the last *SC* (100th in the present implementation) or a constant value to the input ports of the first *SC*. The constant values is set to zero and can be regarded as the row of the dynamic programming matrix at position -1. This constant zero value essentially represents the dynamic programming matrix row that immediately precedes the next row to be calculated.

The *SC* unit (Figure 3) calculates Equation 1 (see Section II). Initially, the B_j value, which represents a 2-bit coding of a nucleotide in the QS, is transformed into the corresponding 4-bit representation by the *NUC_DEC_UN* unit. Then, the *Q_R_CMP_UN* unit performs a comparison between the QS nucleotide and the current RS position. Based on the result of this comparison *and* the CGAP signal, the 4to1 multiplexer selects one out of the four possible values for the intermediate $D^{i-1,j-1} + S^{i,j} + CG^i$ value. The four possible values are either one of the three constants values for a mismatch (MisM, 3), a match with a CGAP (MaCG, 10), or a mismatch with a CGAP (MisM+MaCG, 13), or the value $D^{i-1,j-1}$ which is the output of a neighboring SC.

The three parallel components denoted as *MIN* in Figure 3 are used to select the minimum value among the input signals (see Equation 1 in Section II). In fact, only two *MIN* comparison components would be required. The reason for using an additional *MIN* module is for shortening the critical path and thereby obtain a higher operating clock frequency. At the same time, the latency of the computational part of the SC still amounts to only 1 clock cycle.

Finally, the *VAR_LEN_SHFT_REG* components are variable length ram-based shift registers that increase the latency of the *SCs* relative to the QS length. Since the *SCORING_CELL_ARRAY* comprises 100 *SCs*, scoring requests with a QS length of less than 100 nucleotides do not require additional latency because the processing array can operate on a different site during every clock cycle. For instance, when a matrix for a QS length of 350 nucleotides is computed, the *VAR_LEN_SHFT_REG* registers will increase the latency to 4 clock cycles. The cell values are temporarily buffered in the pipeline stages of the *SCs* until the *SCORING_CELL_ARRAY* finishes the operations at some site i and is able to proceed to site $i+1$. During the 4 clock cycles, the QS is provided as input to the *SC* array in blocks of 100 nucleotides per cycle. For 350 nucleotides, the last 50 *SCs* of the array will receive undefined input data in every 4th clock cycle and the respective outputs will be ignored by the *FINAL_SCORE_DETECT_UNIT*.

V. Implementation, Verification & Design of Accelerator System

We initially described the implementation and verification of the SPU architecture (Section V-A). Thereafter, we describe the PC-FPGA accelerator system as well as the communication protocol between the FPGA board and the host PC (Subsection V-B).

A. Verification of the SPU Architecture

The SPU architecture was implemented in VHDL and initially mapped on a Virtex 5 SX95T-2 FPGA. Extensive post place and route simulations were conducted in order to verify the functionality and correctness of the proposed architecture. As simulation tool, we used Modelsim 6.3f by Mentor Graphics. We used the PaPaRa source code to generate appropriate testbenches for real-world biological datasets.

Thereafter, the HTG-V5-PCIE development board with the same Virtex 5 SX95T FPGA was used for testing on the actual chip. The main objective of these tests was to verify the correctness of the SPU architecture. We used an advanced verification tool (the Chipscope Pro Analyzer) to monitor the input and output ports of the SPU.

B. FPGA-based Accelerator System

After successful verification of the SPU architecture (post place and route simulations *and* tests on an actual FPGA) we integrated a simplified communication protocol between the FPGA board (HTG-V5-PCIE) and the host PC. For this purpose, a Dell Latitude E4300 series laptop with an Intel Core2 Duo P9400 processor (2.4GHz, Ubuntu) was used. The communication between the FPGA board and the PC was established over Gigabit Ethernet using a dedicated UDP/IP core for direct PC-FPGA communication [21]. Figure 5 illustrates the complete system.

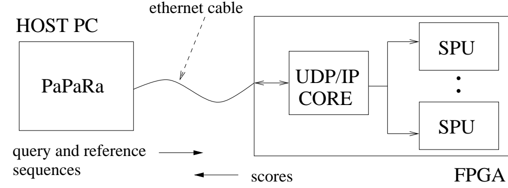

Figure 5. The FPGA-based acceleration system.

A special sequence of packets needs to be transmitted such that, the PaPaRa software (on the PC) can trigger SPU operations. An alignment request requires the transmission of a *Q_PACK*, that is, an ethernet packet that contains the query sequence, followed by an *R_PACK* which contains the reference sequence against which the QS shall be aligned. Every X packets, where X is the size of the *RES_FIFO_BUF*, a *TB_PACK* packet (stands for Transmit Back) is transmitted. The *TB_PACK* packet indicates to the SPU that the software is ready to receive the results from the previous alignment requests that are stored in *RES_FIFO_BUF*. Figure 6 illustrates the supported packet formats.

To further optimize the communication process, we configured the FPGA to transmit and receive JUMBO frames (i.e., IP packets that are longer than the standard minimum length of 1500 bytes). This allowed us to transmit only a single ethernet packet for each alignment request. Thus, a common alignment request now requires the transmission of a new packet type, the *QR_PACK*. Here, the *Q_PACK* format can be concatenated with the *R_PACK* into the same large UDP packet. Also, a modified transmit back request *TB_QR_PACK*, asking for the results of a previous alignment request, can be inserted at the beginning of the common alignment request *QR_PACK* format to replace the stand-alone *TB_PACK* packet.

Each packet format starts with a unique command code. In the query and reference packets, the command code occupies 4 bytes while in the transmit back packet it occupies 2 bytes. In addition, the *Q_PACK* contains one byte with zeros which is ignored by the SPU, a selection code which is used by the *FINAL_SCORE_DETECT_UNIT* to multiplex the results of the 100 parallel *SCs*, a load code which is used to control the variable length shift registers of the *SCs*, and, finally, the query sequence. Accordingly, the *R_PACK* contains a 2-byte field that contains the reference sequence length and thereafter the raw reference sequence data. Finally, in the *TB_PACK* format, the 2-byte TB command is followed by another 2-byte field that contains the total number of results that shall be transmitted back. The number of results can not exceed the size of the *RES_FIFO_BUF* memory.

We created an experimental extension of the PaPaRa program on the PC side. The original code is used for reading input files, encoding QS and RS into the appropriate format, sending SPU-compliant UDP packets to the FPGA,

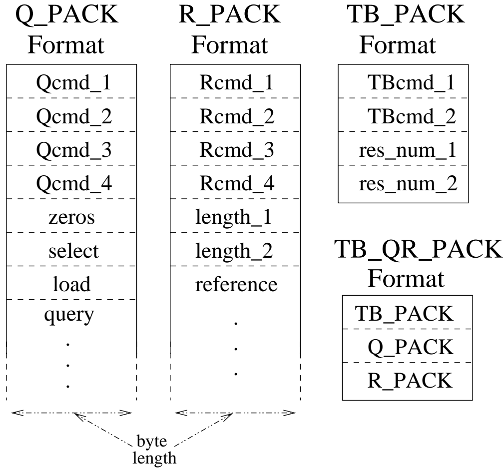

Figure 6. Basic packet formats.

receiving the result packets from the FPGA, performing the trace-back step, and generating the actual QS alignments for the best insertion edges. The program extension was created in C++ using the boost library (www.boost.org) for sending and receiving UDP packets.

The fast response time of the FPGA led to a difficult technical challenge for the PC application when synchronous communication is used. After the jumbo UDP packet containing the QS and RS has been sent (QR_PACK) to the FPGA, the FPGA will send back the resulting scores within a very short amount of time (only 6 cycles after the end-of-frame signal of the UDP packet is set). Because program execution on the PC side is blocked until the whole UDP packet has been sent via a system call, the answering packet may already have been sent by the FPGA, before the PC can actually start receiving it which again requires a system call. Incoming UDP packets are generally not buffered, if no corresponding receive operation has been initiated, that is, the packets returned by the FPGA can be lost. Therefore, we deploy an asynchronous communication mechanism, which relies on a dedicated thread that initiates a receive operation, *before* the QS and RS are sent to the FPGA. We used the asynchronous IO operations as provided by the boost library to implement this functionality.

Note that, the PAR (Place and Route) static timing report revealed that the maximum operating clock frequency of an SPU on a Virtex 5 SX95T FPGA is 101.52 MHz. In order to avoid additional synchronization problems with the software and, at the same time, be able to verify the functionality of the proposed communication protocol, the same clock signal of 125 MHz was used both, for the communication components on the FPGA, and the SPU. By communication components, we refer to the TEMAC (Tri-Mode Ethernet Media Access Controller) *and* the UDP/IP core. This SPU over-clocking caused the incorrect computation of a fraction

Resources	1 SPU V6	12 SPUs V6	1 SPU SYS V5
Slice Regs	7,725	92,248	8,366
Slice LUTs	29,026	347,344	29,103
Occup. Slices	7,791	88,036	9,059
BlockRAMs (36k)	5	60	5
BlockRAMs (18k)	5	60	10
TEMACs	-	-	1

Table I
OCCUPIED RESOURCES ON THE V5SX95T AND V6HX565T DEVICES.

of the scores by the SPU. According to the error report that was generated by the software during the final alignment phase, the fraction of incorrect scores due to over-clocking the prototype system ranged between 10% and 20% of overall SPU score requests.

VI. RESULTS

Here we present a performance assessment for our accelerator architecture when multiple SPUs are instantiated on a large Virtex 6 FPGA. All results presented in the current Section refer to Xilinx reports after the implementation process (post place and route).

Table I provides resource usage reports for three hardware configurations. The first column of Table I provides the resources occupied by a single SPU instance on a Virtex 6 HX565T-2 FPGA (`1_SPU_V6`). According to the PAR static timing report, the maximum operating frequency for this unit is 140.92 MHz. The 12_SPU_V6 architecture comprises 12 independent SPU instances with a maximum operating frequency of 111.17 MHz. Finally, the `1_SPU_V5_SYS` implementation, which was described in the previous Section, contains 1 SPU instance, a TEMAC, and a UDP/IP core.

For performance comparison we executed the software-only implementation of PaPaRa on an Intel core i5 750 CPU running at 3.2 GHz. Initially, we measured the total execution time for the scoring phase (aligning all QS against all ancestral states) required by the PaPaRa implementation for 4 real-world biological datasets. The PaPaRa software contains a special algorithmic optimization (the so called 'early stopping criterion'), which decreases the total number of matrix cells that need to be calculated. This trick improves the runtime of the PaPaRa software by a factor of 2–3 [2]. The current FPGA design does not implement this optimization, since this would require a non-trivial re-design of the pipeline datapath for calculating the matrix entries *and* of the communication protocol.

Nonetheless, to conduct a fair comparison, we present performance data for the standard software implementation as well as for the optimized software implementation with the early stopping criterion. For a given dataset, the standard software implementation performs exactly the same number

Dataset	Alignment kernel PC (base)	Alignment kernel PC (opt)	Alignment kernel FPGA	Trace back	Kernel Speed Up VS PC (base)	Kernel Speed Up VS PC (opt)	Application Speed Up VS PC (base)	Application Speed Up VS PC (opt)
D218_200	1,359	841	1.82	9.6	748.8	463.7	119.9	74.6
D218_500	2,868	1,890	4.01	19.9	715.2	471.2	120.8	79.9
D500_200	4,125	1,772	6.4	13	641.3	275.5	213.0	91.9
D500_500	7,872	3,784	14.18	22.9	555.1	266.8	212.9	102.7
D855_200	12,516	4,269	19.37	23.6	646.2	220.4	291.8	100.0
D855_500	23,947	9,604	42.69	41.6	560.8	224.9	284.6	114.4
D1604_200	38,333	12,109	60.50	42.5	633.6	200.1	372.6	118.0
D1604_500	69,815	22,684	133.26	68.8	523.9	170.2	345.9	112.6

Table II
TOTAL EXECUTION TIMES (IN SECONDS) OF THE ALIGNMENT KERNEL AND THE TRACE-BACK STEP OF THE PAPARA ALGORITHM AND THE RESPECTIVE SPEED UPS OF THE FPGA SYSTEM FOR THE ALIGNMENT KERNEL AND FOR THE COMPLETE APPLICATION. THE STANDARD ALGORITHM IMPLEMENTATION IS DENOTED AS *base* WHILE THE OPTIMIZED VERSION IS DENOTED AS *opt*.

of cell updates as the current hardware design. The second phase of PaPaRa ('alignment phase') is always executed on the PC for the software-based FPGA-accelerated implementation. Table II provides execution times and the speedups that can be achieved by offloading the alignment kernel to a Virtex 6 FPGA that contains 12 SPUs. The input dataset names in the left column denote the number of taxa in the original biological dataset followed by the average QS length (see [2]).

The required data transfer rate for a SPU-based accelerator system is given by the following formula:

$$I = [(SPU_N * SC_N * Q_CD) + R_CD]/CLK_P$$

where `SPU_N` is the number of SPUs in the design (12 in our implementation), `SC_N` is the number of scoring cells in the processing array of each SPU (100 in our implementation), `Q_CD` and `R_CD` is the number of bits required for representing a QS character and a RS ancestral state for one site (2 and 5 in our implementation), and finally `CLK_P` is the clock period. Based on the above formula, for the given configuration, data has to be provided to the SPUs at a rate of 31.2 Gb/s in order to achieve the maximum possible speedups reported in Table II. While this transfer rate is higher than what we can currently achieve with existing PC-FPGA communication methods like Gigabit Ethernet (max. 125MB/s) or PCI Express (max. 16 GB/s), the requirement can be met by using block ram memory based input/output buffers for storing input data. Ideally, (i.e., if the input data fits into the buffer), each query and reference sequence will have to be transferred to the respective buffer only once and can then be fed into the SPUs several times. Clearly, the efficiency of this approach depends on an appropriate input/output buffer size and a suitable buffer management. Note that, the communication overhead for transmitting the results produced by the SPUs (i.e., the final matrix scores) back to the PC can be neglected. In contrast to input data transfers, every SPU only generates a 16-bit alignment score value every INTERVAL(Q_CD) * R_CD clock cycles. The INTERVAL (Query length) function returns the number of clock cycles spent by the *SCORING_CELL_ARRAY* for each column/site of the matrix (each ancestral vector state).

VII. CONCLUSIONS & FUTURE WORK

We presented a hardware/software implementation for boosting performance of a novel short read alignment method, that simultaneously aligns reads to reference multiple sequence alignments and corresponding phylogenetic trees. The software and hardware implementations are available for download as open source code. The reconfigurable architecture was verified on an actual Virtex 5 FPGA and the functionality of the communication protocol was tested using gigabit ethernet. The hardware architecture achieved speedups for the score calculation phase of PaPaRa ranging between 170 and 471 on a Xilinx Virtex 6 FPGA compared to the most efficient software version of PaPaRa on an Intel Core i5 CPU running at 3.2GHz. To the best of our knowledge, this represents the first FPGA-based accelerator architecture for this novel alignment kernel.

With respect to future work, we plan to initially improve the computational pipeline datapath. Additional pipeline stages will be introduced to allow for a higher maximum SPU operating clock frequency. Furthermore, we plan to adapt the communication protocol for eliminating input/output delays. The current communication protocol does not represent the ideal solution, since the synchronization between the host PC and the SPUs is achieved by consecutive retransmissions of the same reference ancestral state sequence. Thus, we plan to design an improved software/hardware implementation that can pre-load the input files into the memory on the FPGA or the external memory on the board. A dedicated I/O controller comprising a set of memory buffers will be used for hiding communication latency from the SPUs and thereby allow them to operate

at maximum speed. If gigabit ethernet is not sufficient for achieving this, we will consider a solution using PCI Express.

Another direction of future work is to optimize the number of parallel scoring cells in a processing unit. The fixed number of 100 scoring cells in the current proof-of-concept implementation was merely chosen for verification purposes. We plan on conducting performance tests using the PaPaRa software and real-world biological data to determine the optimal number of SPUs across a wide range of datasets. Finally, we will investigate if alternative accelerator technologies, such as GPUs, can be used to achieve comparable speedups or even outperform FPGAs for this specific type of application.

REFERENCES

[1] J. Felsenstein, "Evolutionary trees from DNA sequences: a maximum likelihood approach," *J. Mol. Evol.*, vol. 17, pp. 368–376, 1981.

[2] S. Berger and A. Stamatakis, "Aligning short Reads to Reference Alignments and Reference Trees," Heidelberg Institute for Theoretical Studies, Tech. Rep. Exelixis-RRDR-2010-9, 2010, http://wwwkramer.in.tum.de/exelixis/publications.html.

[3] F. Matsen, R. Kodner, and E. V. Armbrust, "pplacer: linear time maximum-likelihood and bayesian phylogenetic placement of sequences onto a fixed reference tree," *BMC Bioinformatics*, vol. 11, no. 1, p. 538, 2010. [Online]. Available: http://www.biomedcentral.com/1471-2105/11/538

[4] M. Stark, S. A. Berger, A. Stamatakis, and C. von Mering, "MLTreeMap - accurate Maximum Likelihood placement of environmental DNA sequences into taxonomic and functional reference phylogenies," *BMC Genomics*, vol. 11, no. 1, pp. 461+, August 2010. [Online]. Available: http://dx.doi.org/10.1186/1471-2164-11-461

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

[6] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, pp. 195–197, 1981.

[7] Gotoh, "An improved algorithm for matching biological sequences," *J. Mol. Biol.*, vol. 162, pp. 705–708, 1982.

[8] A. Stamatakis, "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinformatics*, vol. 22, no. 21, pp. 2688–2690, 2006.

[9] N. Alachiotis, A. Stamatakis, E. Sotiriades, and A. Dollas, "A reconfigurable architecture for the phylogenetic likelihood function," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 31 2009-sept. 2 2009, pp. 674 –678.

[10] A. Dollas, "Reconfigurable architectures for bioinformatics applications," *VLSI, IEEE Computer Society Annual Symposium on*, vol. 0, pp. 6–7, 2010.

[11] D. Sankoff, "Minimal mutation trees of sequences," *SIAM. J. Appl. Math.*, vol. 28, pp. 35–42, 1975.

[12] I. Li, W. Shum, and K. Truong, "160-fold acceleration of the smith-waterman algorithm using field programmable gate array (fpga)," *BMC Bioinformatics*, vol. 8, no. 1, p. 185, 2007. [Online]. Available: http://www.biomedcentral.com/1471-2105/8/185

[13] C. Yu, K. Kwong, K. Lee, and P. Leong, "A smith-waterman systolic cell," in *New Algorithms, Architectures and Applications for Reconfigurable Computing*, P. Lysaght and W. Rosenstiel, Eds. Springer US, 2005, pp. 291–300.

[14] X. Jiang, X. Liu, L. Xu, P. Zhang, and N. Sun, "A reconfigurable accelerator for smith-waterman algorithm," in *IEEE Trans. Circuits and Systems II*, 2007, pp. 1077–1081.

[15] B. Harris, A. Jacob, J. Lancaster, J. Buhler, and R. Chamberlain, "A banded smith-waterman fpga accelerator for mercury blastp," in *Proceedings of International Conference on Field Programmable Logic and Applications 2007*, ser. FPL '07, 2007, pp. 765–769.

[16] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[17] Y. Liu, D. Maskell, and B. Schmidt, "Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units," *BMC Research Notes*, vol. 2, no. 1, p. 73, 2009. [Online]. Available: http://www.biomedcentral.com/1756-0500/2/73

[18] J. F. Eusse Giraldo, N. Moreano, R. P. Jacobi, and A. C. M. A. de Melo, "A hmmer hardware accelerator using divergences," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10, 2010, pp. 405–410. [Online]. Available: http://portal.acm.org/citation.cfm?id=1870926.1871023

[19] R. C. Edgar, "Muscle: multiple sequence alignment with high accuracy and high throughput," *Nucleic Acids Research*, vol. 32, no. 5, pp. 1792–1797, 2004.

[20] K. Katoh and H. Toh, "Recent developments in the MAFFT multiple sequence alignment program," *Briefings in Bioinformatics*, vol. 9, no. 4, pp. 286–298, 2008.

[21] N. Alachiotis, S. A. Berger, and A. Stamatakis, "Efficient pc-fpga communication over gigabit ethernet," in *CIT*, 2010, pp. 1727–1734.

[22] K. Benkrid, Y. Liu, and A. Benkrid, "A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment," *IEEE Transactions on VLSI Systems*, vol. 17, pp. 561–570, 2009.

[23] J. Allred, J. Coyne, W. Lynch, V. Natoli, J. Grecco, and J. Morrissette, "Smith-waterman implementation on a fsb-fpga module using the intel accelerator abstraction layer," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–4, 2009.