An Optimized Reconfigurable System for Computing the Phylogenetic Likelihood on DNA Data

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

***Abstract*—The Phylogenetic Likelihood Function (PLF) is an important statistical function for evaluating phylogenetic trees. To this end, the PLF is *the* computational kernel of all state-of-the-art likelihood-based phylogenetic inference programs. Typically, it accounts for more than 85% of total execution time in such programs. We present a substantially improved hardware architecture for computing the PLF based on previous experiences with implementing the PLF on reconfigurable logic. Our new design is optimized for computing the PLF on four-state (DNA) input data. It is also adapted to the computational requirements of real-world tree inference programs and completely independent of the specific tree search algorithm at hand. Furthermore, we describe how our architecture can be modified and adapted to handle general n-state data, such as protein (20 states) or RNA secondary structure data (6, 7, or 16 states, depending on the model). Finally, we designed an interface mechanism such that our PLF hardware architecture can interact with the widely-used phylogenetic inference tool RAxML. We deploy FPGA technology to verify the correctness of the architecture and to evaluate performance.**

I. Introduction

Phylogenetic inference is a discipline of computational biology that deals with reconstructing trees (phylogenies) that represent the evolutionary history of a set of species. Inferences can, for instance, be based on molecular sequence data. Phylogenetic tree reconstruction has many important applications in biological and medical research (see e.g., [1]).

The input for a phylogenetic analysis consists of a list of organism (species) names and their respective DNA sequence data. Prior to conducting a phylogenetic analysis using character-based inference methods (e.g., Maximum Parsimony [2] or Maximum Likelihood [3]), a multiple sequence alignment (MSA) of the sequences needs to be computed in a pre-processing step. The goal of the MSA step is to determine which nucleotides of the sequences share a common evolutionary history. An exemplary MSA of DNA sequences for the Human, the Mouse, the Cow, and the Chicken is provided below:

```
Cow       ATGGCATATCCCA-ACAACTAGGTCCAA
Chicken   ATGGCCAACCACTCCCAACTAGGTTC-A
Human     ATGGCACAT---GCGCAAGTAGGTAC-A
Mouse     ATGG----CCCATTCCAACTTGGTACAA
```

The gap symbol - is inserted into the sequences by the MSA process to indicate that nucleotide insertions or deletions have occurred during the evolutionary history of the organisms under study. After the MSA step, all n sequences have the same length m, that is, the MSA has m alignment columns (also called sites).

Given a MSA, the output of a phylogenetic analysis is an *unrooted* binary tree topology. The extant species (for which DNA data *can* be sequenced and *is* available) are assigned to the leaves (tips) of such a tree, whereas the inner nodes represent hypothetical extinct common ancestors.

In general, a scoring function/criterion and a tree search strategy are required to reconstruct a phylogeny from a MSA. Finding the best-scoring tree under Maximum Likelihood is known to be NP-hard [4]. Intuitively, this is because the number of possible distinct unrooted binary tree topologies for n organisms is super-exponential in n ($\prod_{i=3}^{n}(2i-5)$). The scoring criteria/functions are used to assess how well a specific tree topology explains (fits) the underlying molecular sequence data.

The PLF represents one of the most widely used optimality criteria to score and thus choose among distinct evolutionary scenarios (phylogenetic trees). Numerous phylogenetic inference packages implement the PLF, either for standard ML-based optimization (RAxML [5], GARLI [6], PHYML [7]) or for Bayesian phylogenetic inference (MrBayes [8], PhyloBayes [9]). All PLF-based phylogenetic inference programs spend the largest fraction of overall run time (typically between 85% and 95% [10]) for computing the PLF. It is thus desirable to devise hardware solutions for this widely-used and important function.

The work we present here, is already the 4th generation of our series of reconfigurable architectures for computing the PLF. Initially, we explored two alternative approaches (1st generation:[11] and 2nd generation:[12]) for parallelizing the PLF on hardware. We found that, a deeply pipelined tree-like placement of likelihood processing units (1st generation design) can compute the PLF on fully balanced binary trees in a fast and memory-efficient way. However, a more flexible vector-like arrangement of likelihood processing units (2nd generation design) is more suitable for real-world scenarios, since it is completely independent of the tree-search strategy, the size, as well as the shape of the tree. After adopting this generic vector-like arrangement of processing units for the PLF, we designed the 3rd generation architecture [13] that provided the full functionality for offloading all PLF functions from

a real-world Maximum Likelihood program (RAxML) to a co-processor. The 3rd generation design supports DNA, AA, and RNA secondary structure data, scaling procedures for avoiding numerical underflow, Newton-Raphson-based branch length optimization, and the calculation of transition probability matrices (see Section III). We realized that, such a highly flexible and generic architecture (capable of executing several PLF function types and handling data with different numbers of states) can not be efficiently mapped (compared to performance on x86 architectures) onto present-day FPGAs.

The 4th generation of our architecture, which we present here, combines the concepts developed for the 2nd and 3rd generation and has been significantly improved with respect to resource utilization as well as performance. In addition, our new architecture implements a generic, FIFO-based interface for communicating with the outer world via, for instance, external memory or PCIExpress.

The remainder of this paper is organized as follows: in Section II we address related work and in Section III we describe how the PLF is calculated. In Section IV we outline the architecture and in Section V we describe how the software is (re-)organized to allow the RAxML application code (running on the CPU) to efficiently offload operations to the hardware architecture. We present performance results in Section VI-B and conclude in Section VII.

II. Related Work

While there exists a large diversity of methods and software for phylogenetic inference, to date, only few methods have been mapped to hardware. In [14] and [15], Mak and Lam map a PLF implementation with reduced floating-point precision to reconfigurable logic. The Jukes-Cantor (JC69 [16]) model, which is implemented in this work, represents the simplest statistical model of DNA substitution and, as a consequence, is rarely used in present-day biological analyses [17]. The performance tests reported in [14] and [15] have been conducted on trees with only 4 leaves (4 input sequences). Hence, scalability beyond 4 species trees is not addressed.

In [18], Davis *et al.* present an implementation of a simple tree reconstruction method called UPGMA (Unweighted Pair Group Method with Arithmetic Mean). Due to the many simplifying assumptions made in the UPGMA algorithm, it is practically not used for real-world analyses any more.

In [19] and [20], Bakos *et al.* focus on the reconstruction of phylogenetic trees using gene order input data, that is, the order of corresponding genes in the genomes of different organisms is used as input data for reconstructing trees. Bakos *et al.* mapped GRAPPA [21], an open-source implementation for gene order based phylogenetic inference, onto FPGAs. The main difference to PLF-based phylogenetic inference is that, the kernel function used in gene order analyses is discrete. This means that, the amount of floating-point operations required to reconstruct a phylogeny is small and that a FPGA implementation can mostly rely on integer arithmetics.

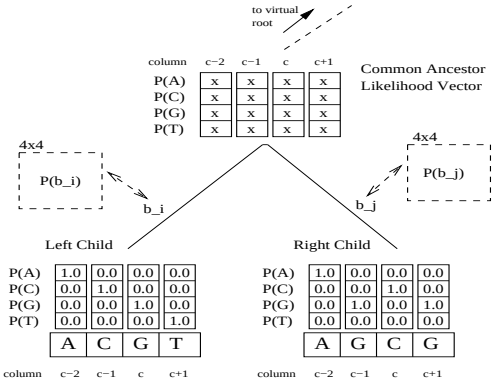

Figure 1. Computation of ancestral probability vector entries.

Zierke and Bakos [22] presented a FPGA accelerator for the PLF as required for Bayesian MCMC-based (Markov Chain Monte Carlo) inference methods. The authors mapped the MrBayes [8] PLF implementation for DNA data to reconfigurable hardware. They included numerical scaling techniques (see [23] for details on numerical scaling in the PLF) to prevent underflow as well as a component for calculating log likelihood scores. The speedup estimates (based on the largest available Virtex 6 SX FPGA at time of publication) that are reported in the paper vary between 2.5 and 8.7 compared to a single state-of-the-art Intel Xeon 5500-series core. Note that, Bayesian inference programs do not require numerical optimization routines (e.g., Newton-Raphson) for branch length optimization, since the MCMC procedure is used to integrate over branch length distributions. Hence, the PLF as used in Bayesian inference programs is less complex than for ML programs. ML programs typically deploy Newton-Raphson optimization procedures for branch length optimization, which makes hardware design more challenging, since dedicated components for computing the first and second derivative of the likelihood are required.

Kasap and Benkrid [24] presented a FPGA design for phylogenetic inference under parsimony and assessed performance on a FPGA supercomputer. The implementation is limited to trees with up to 12 species, which is very small by today's standards (the largest published parsimony-based tree has 73,060 taxa [25]). They report speedups between a factor of 5 and up to a factor of 32,414 for utilizing 1, 2, 4, and 8 nodes (each node is equipped with a Xilinx Virtex4 FX100 FPGA) on the Maxwell system compared to a 2.2GHz Intel Centrino Duo processor. However, the speedups reported are only relative speedups with respect to the parsimony function implementation in PAUP* [26] and not with respect to the fastest-known parsimony implementation in the TNT software package (used, e.g., in [25]).

In [27], Alachiotis and Stamatakis mapped a generic version of the parsimony kernel to reconfigurable logic. The implementation is independent of the number of species and is adapted to the requirements of modern tree search strategies. They verified the functionality of the architecture using real-world datasets with up to $n = 500$

species and $m = 48,000$ alignment sites. The speedups ranged between a factor of 5.6 and 12.5. The parsimony architecture was mapped to a Xilinx Virtex 6 SX475 FPGA and performance was compared to the highly optimized version of the parsimony kernel in the Parsimonator open-source code (available at: http://www.exelixis-lab.org/software.html; Parsimonator uses SSE3 and AVX vector intrinsics) running on a Intel i7 2600 CPU at 3.4GHz.

III. The Phylogenetic Likelihood Function

The Felsenstein pruning algorithm [3] is the standard method for computing the PLF and the overall log likelihood score on a given tree topology. In the following, we provide an abstract description of this algorithm.

The first step consists in identifying a pair of child nodes i and j in the given tree for which the probability vector at the common ancestor k ($1 \leq i, j, k \leq 2n-2$) has not yet been computed. Given nodes i and j, the second step consists in calculating the ancestral probability vector entries at k and to subsequently prune the child nodes i and j from the tree. These steps are executed recursively (essentially by means of a post-order tree traversal) until the probability vector at the virtual root vr has been calculated. When this is the case, the pruning process has transformed the initial tree to only one node (ancestral probability vector) that is located at the virtual root of the tree. Phylogenetic trees under ML are unrooted for mathematical and computational reasons [3], but a virtual root vr can be placed into any branch of the tree to evaluate its likelihood score. Note that, under certain, standard, model restrictions (namely time reversibility of the model of nucleotide substitution) the overall log likelihood score is identical regardless of the placement of the virtual root.

In order to compute the PLF on a given, fixed, tree topology, apart from the tree shape itself, one also requires the branch lengths and the parameters of the statistical nucleotide substitution model (see [11] and [23] for details). For DNA data, a model of nucleotide substitution is provided by a 4×4 matrix (for protein data by a 20×20 matrix) that is usually denoted as Q matrix. The Q matrix contains the instantaneous transition probabilities (for infinitesimal relative evolutionary time dt) of a nucleotide A to change into a nucleotide A, C, G, or T etc. To compute the nucleotide substitution probabilities for a given branch length t (t essentially represents the evolutionary time between two nodes in the tree), one has to compute $P(t) = e^{Qt}$. This is usually implemented via an eigenvalue/eigenvector decomposition. Thus, to compute the likelihood on a fixed tree with given branch lengths and model parameters, one initially needs to apply the Felsenstein pruning algorithm and subsequently compute the overall likelihood score of the tree based on the ancestral probability vector at the virtual root.

Every probability vector entry at a position c ($c = 1...m$) $\vec{L}(c)$ at the tips and at the inner nodes contains the four probabilities $P(A)$, $P(C)$, $P(G)$, $P(T)$ of observing a nucleotide A, C, G, or T at a specific column/site c of the input alignment. The probabilities at the tips (leaves) of the tree for which observed data (DNA sequences) *is* available are set to 1.0 for the observed nucleotide character at the respective position c (e.g., for the nucleotide A: $\vec{L}(c) = (1.0, 0.0, 0.0, 0.0)$).

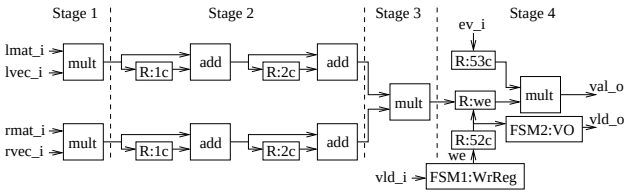

Figure 2. Block diagram of the pipelined PLF architecture for DNA data.

Given, a parent node k and two child nodes i and j, their probability vectors $\vec{L}^{(i)}$ and $\vec{L}^{(j)}$, the respective branch lengths leading to the children b_i and b_j, and the transition probability matrices $P(b_i), P(b_j)$, the probability of observing an A at position c of the ancestral (parent) vector $\vec{L}_A^{(k)}(c)$ is computed as follows:

$$\vec{L}_A^{(k)}(c) = \Big(\sum_{S=A}^{T} P_{AS}(b_i) \vec{L}_S^{(i)}(c) \Big) \Big(\sum_{S=A}^{T} P_{AS}(b_j) \vec{L}_S^{(j)}(c) \Big) \quad (1)$$

A schematic outline of this procedure is provided in Figure 1. As mentioned before, an important practical implementation issue is that, the ancestral probability vector entries need to be scaled for avoiding numerical underflow because of increasingly small probability values as we approach the virtual root.

When the procedure reaches the virtual root, the per-site likelihood at site/column c, $l(c)$ is computed as follows using the probability vector $\vec{L}^{(vr)}$ at the virtual root:

$$l(c) = \sum_{S=A}^{T} \pi_S L_S{}^{(vr)}(c) \quad (2)$$

The probabilities π_A through π_T are the prior probabilities (also called base frequencies) of observing A, C, G, or T at vr and are typically drawn empirically from the input data. The overall likelihood of the tree is then computed as the sum over the logarithm of the per-site likelihoods: $\sum_{c=1}^{m} log(l(c))$.

IV. Reconfigurable Architecture

In the following, we describe the basic pipelined datapath of the PLF, the complete likelihood core as adapted for DNA data, the scaling unit, and the FIFO-based interface to the "outer" world.

A. PLF Pipeline

Figure 2 illustrates our pipelined datapath. There are four 64-bit wide input buses that are used for streaming in the input data. We obtain the left and right transition probability matrices ($P(b_i), P(b_j)$) over the *lmat_i* and *rmat_i* buses. The probability vector entries of the left and right probability vectors are streamed in ($\vec{L}^{(i)}$, $\vec{L}^{(j)}$) via the *lvec_i* and *rvec_i* buses. The pipeline is organized into

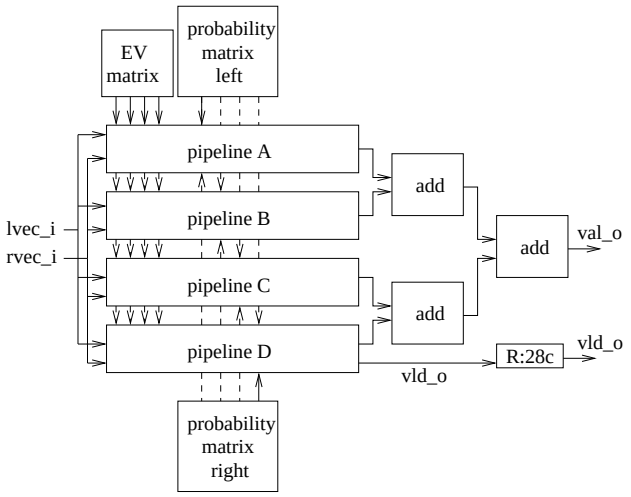

Figure 3. Block diagram of the likelihood core with 4 instances of the PLF pipeline.

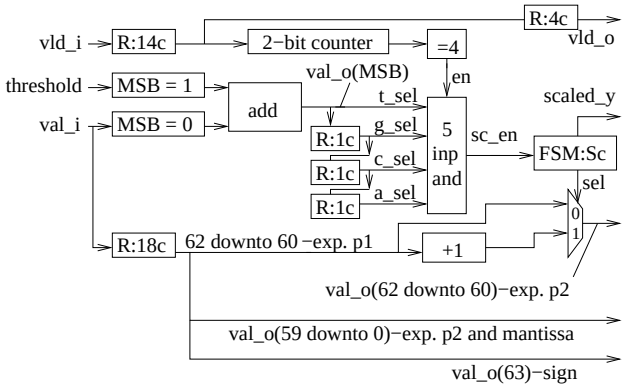

Figure 4. Architecture of the pipelined ancestral vector entry scaling unit.

4 main stages which are distinguished by dashed lines in Figure 2. **Stage 1** consists of 2 pipelined multipliers that operate in parallel and compute the sum of the 2 input arguments from the left and right child node. Thereafter, **Stage 2** accumulates the multiplier outputs within four clock cycles (remember: DNA data has 4 states). The results of the two parallel accumulation operations are combined in **Stage 3** via a multiplication. The combination of stages 1, 2, and 3 implements Equation 1. Finally, **Stage 4** executes the multiplication with the inverted eigenvector matrix (this is a numerical detail that is related to the exponentiation of the Q matrix as described in Section III). Thus, a single pipeline instance can perform all operations required to multiply a column of a transition probability matrix P with a probability vector entry $\vec{L}(c)$.

B. Likelihood Core

The complete likelihood core for DNA data is shown in Figure 3. Four pipeline instances are required to accommodate the 4-state DNA model. Every pipeline instance operates on a different column of the two 4×4 probability transition matrices ($P(b_i), P(b_j)$) that represent the branch lengths leading to the left and right child node respectively. All instances receive the same probability vector input and the same inverted eigenvector matrix. Note that, only a single Q matrix is used for calculating the PLF on the entire tree. Thus, the eigenvector matrix for obtaining $P(b_i)$ and $P(b_j)$ remains constant. The results of the individual pipeline instances are then summed up using the adder tree depicted in Figure 3. The *val_o* output bus is used to stream out the ancestral probability vector entry $\vec{L}^{(k)}$.

C. Scaling Unit

As already mentioned, the output values $\vec{L}^{(k)}$ of the likelihood core need to be checked for potential numerical underflow and scaled appropriately, if required (for mathematical details, please refer to [23]). Therefore, we designed a pipelined probability vector entry scaling unit (Figure 4). The *scaler* unit works as follows. It monitors the values that are generated by the likelihood core on a per-site basis and scales/multiplies them by a constant factor if all values in an ancestral probability vector entry $\vec{L}^{(k)}(c)$ are smaller than a pre-defined threshold ϵ.

Initially, an adder is used to compare the incoming probability values to ϵ. This is achieved by adding the negative value of the threshold to the input probabilities that need to be checked for scaling. The sign bit of the adder output indicates whether the values need to be scaled or not. Since we only scale, if *all* four probabilities of an output vector are smaller than ϵ, we use three 1-bit shift registers with a delay of one clock cycle. Then, a five-way *and* operation is used to determine the value of the *sc_en* signal. Hence, the *sc_en* is used to decide if we need to scale or not. If we need to scale, we multiply all four probabilities in $\vec{L}^{(k)}(c)$ by 2^{256}. We implement this multiplication via a 3-bit addition on the three most significant bits of the mantissa.

D. System Design

Figure 5 illustrates how the likelihood core and the scaler unit are connected with each other. The figure also shows how the core components are integrated with the FIFO input/output buffers that allow for efficient communication with the outer world (e.g., an external memory controller or a PCIExpress bus). Incoming data are temporarily stored in the input FIFO buffers until enough data have arrived for computing an ancestral probability vector entry at a site c. These dedicated FIFO buffers that are specifically adapted to the operation of our pipeline allow for simplifying the pipeline architecture and thereby obtaining a more resource-efficient design with improved performance.

An important aspect of the proposed architecture is that, it can easily be adapted for accommodating PLF computations on data with more states (e.g., protein data with 20 states or RNA secondary structure data models with 6,7, or 16 states). The required modifications are straight-forward, since one only needs to appropriately adapt the adder/accumulator parts of the PLF pipeline, the likelihood core, and the number of shift registers in

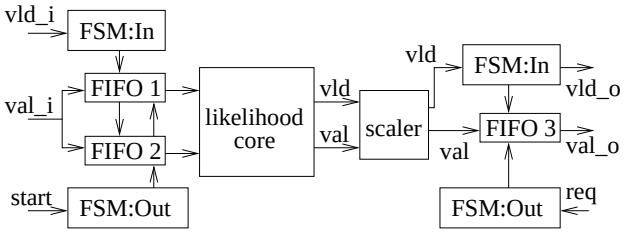

Figure 5. Top-level architecture of the PLF system.

the scaler. More specifically, the current configuration of pipeline **Stage 2** accumulates input values over four clock cycles. Extending this stage by more shift-register/adder pairs, allows for deploying the pipeline on data with more states. A different number of pipeline stages is required in the likelihood core as well as for the adder-tree. The number of shift-registers in the scaler component must be increased to match the number of states minus one. Also the size of the *and* gate needs to be adapted.

V. Host-side management

In the following, we describe the required software-side components required for using our FPGA-based system. To design an efficient hardware implementation, we retain the entire "logic" that offloads *to* and steers operations *on* the hardware, on the PC-side. Thereby, we can deploy the hardware architecture to exclusively carry out the expensive PLF computations. Therefore, the host (PC-side) performs two main system tasks: (i) manage memory and (ii) initiate computations on the FPGA.

A. Memory-management

The external memory (typically DRAM) on the FPGA board is entirely managed by the host. To this end, we developed an abstract interface that allows for conveniently managing vectors of fixed size. This interface organizes (keeps track of the organization of) the on-board external memory into an array of fixed-size memory blocks. The size of each memory block exactly corresponds to the size of one full-length probability vector $\vec{L}$. Each individual memory block (probability vector) is associated with a unified vector address that contains a slot number and a bank number. The mapping of a slot number to a physical address in the external memory *on* the FPGA board is implementation-dependent. Assuming a linear, byte-addressable memory space (which for example is used on the host to mirror the content of the external memory on the FPGA board), the physical address of a slot number Idx_{slot} is $Addr_{phys} = Addr_{base} + Idx_{slot} * Size_{block}$, where $Addr_{base}$ is the start address of the memory region used for block storage and $Size_{block}$ is the size (in # bytes) of each block. On the FPGA, the data-organization is more complex than for a linear address space, since one has to distribute each vector over multiple FPGA-side memory blocks to achieve high memory throughput. The host-side interface hides the intrinsic complexity of dividing the vectors into fragments and storing them in different FPGA-side memory blocks. The bank number, that is, the second part of the unified vector-address, is used to distribute data blocks among different external memories on the FPGA board interfaces. This allows for accessing data blocks that reside in different banks in parallel. In the PLF context, reading and writing data in parallel through different memory interfaces is crucial to achieve maximum performance because the contents of two probability child vectors ($\vec{L}^{(i)}$, $\vec{L}^{(j)}$) can be read in parallel. Therefore, the data blocks associated with the two probability vectors of nodes i and j must be kept on different banks that are in turn, associated with distinct memory interfaces.

The host interface implements functions for data block transfer between the host DRAM and the FPGA. To achieve this in practice, we augmented the internal RAxML data-structures by unified probability vector addresses.

B. Offloading Functions to the FPGA

Apart from handling the entire memory management, the host also orchestrates the PLF operations that are executed on the FPGA. As mentioned above, the FPGA implements the PLF, which represents a key component of the RAxML tree search algorithm. One basic component of the tree-search is tree evaluation (computation of the overall log likelihood score of a tree topology) which, as a prerequisite, requires to compute/update the ancestral probability vectors at the inner nodes of the tree via the Felsenstein pruning algorithm. These updates of ancestral probability vectors which are required when the tree topology is changed (in the course of the tree search) account for *the* computationally most intensive part of the PLF in RAxML (approximately 60-70% of total run time). Therefore, we entirely offload these computations to the FPGA.

To offload the computations, we need to transfer the constant (because the DNA sequence is known) probability vectors located at the tips to their corresponding memory locations in the on-board external memory. Thereafter, we traverse the tree from the tips toward the virtual root to calculate the ancestral probability vectors in post-order. To this end, we represent each post-order tree traversal by a traversal descriptor (an ordered list of tree node triplets: (i, j, k) representing the traversal) which is initially transferred to the FPGA. More specifically, each element in the traversal descriptor, contains a triplet of unified vector addresses (corresponding to the address of the parent node k and the two child probability vectors i and j) and a pair of branch lengths (b_i, b_j). The FPGA then executes the PLF as specified by the elements contained in the traversal descriptor *in sequence* until the PLF for all triplets in the list has been computed. For each traversal descriptor element, the reconfigurable architecture reads the data of the two child vectors and writes the result to the parent vector.

C. Host-side simulation

To test the memory-management concept and the FPGA control "logic" we simulated the memory-management

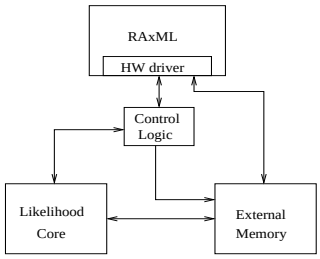

Figure 6. Conceptual design of the complete SW/HW system.

interface and implemented the PLF on the host in SW. We created a "mock-up" that implements the FPGA memory interface. Instead of transferring data between memories on the host and the FPGA, the mock-up emulates the FPGA memory content and the target FPGA memory organization (using 2 banks consisting of 4 internal blocks each) on a x86 architecture. Using this SW emulator, we tested a basic version of the FPGA control logic. Our experiments focused on transferring the probability vectors associated with the tips of the tree to the emulated FPGA memory and testing the iterations over the traversal descriptor (see Section V-B). In our simulations, the PLF was computed on the FPGA emulation (running on the host) and then transferred back from the emulation via the interface to the associated host-side destination address in RAxML. Note that, for these simulations, the address spaces of RAxML and the SW simulation of the FPGA were completely separated. After completing the iteration over the traversal descriptor, we compared the probability vectors at the virtual root produced by the emulated FPGA to the corresponding vectors as calculated directly by RAxML. Thereby, we verified the correctness of the unified vector-addresses storage scheme in RAxML and tested the functionality of the memory allocation strategy.

D. System Overview

A schematic outline of the overall integrated system is provided in Figure 6. RAxML interacts with the reconfigurable hardware via the dedicated HW interface. On the FPGA side, a *Control Logic* module organizes all the incoming processing or data transfer requests that received from the SW on the host side. The RAxML hardware interface can also directly access the external on-board memory to allow for overlapping data transfers with PLF computations. The Control Logic module processes the received traversal descriptor and generates corresponding read/write addresses for the external memory on the FPGA board. The PLF calculations are carried out by the likelihood core and the ancestral state vectors are written back to external on-board memory. The Control Logic synchronizes the likelihood core calculations with read/write operations to external memory. So far, we have implemented the basic RAxML interface as well as an initial test-bench to generate and verify traversal descriptors in software. On the FPGA side, we have implemented, evaluated, and verified the likelihood core.

The hardware description of the reconfigurable system is available for download at: http://www.exelixis-lab.org/countLiCoGen4.php.

Table I
RESOURCE OCCUPATION AND PERFORMANCE OF THE PLF ARCHITECTURE ON A VIRTEX 6 SX475T-2 FPGA.

Resources	1 Core	8 Cores	Maximum
Slice Registers	28,385	226,977	595,200
Slice LUTs	19,596	159,263	297,600
Occupied Slices	7,823	55,692	74,400
DSP48Es	235	1880	2016

VI. VERIFICATION AND RESULTS

We describe the verification of the reconfigurable architecture in Section VI-A and present a performance evaluation of the reconfigurable system in Section VI-B.

A. Verification of the PLF Architecture

Our reconfigurable architecture was implemented in VHDL and mapped to a Virtex 6 SX475T-2 FPGA. We verified the correctness of the hardware design by extensive post place and route simulations using Modelsim 6.3f by Mentor Graphics. We generated input data for the simulations by using RAxML on real-world DNA sequences. In the future, we also intend to conduct tests on an actual chip using a HTG-V5-PCIE development board with a Virtex 5 SX95T FPGA and integrate our open-source PC $\leftrightarrow$ FPGA communication platform [28] (available at http://opencores.org/project,pc_fpga_com) into the RAxML source code for a real-world tests of the entire system.

B. Performance Evaluation

Table I shows the resource requirements for a single instance of the likelihood core on a Virtex 6 SX475-2 FPGA. A direct comparison to previous generations of our architecture can be misleading, since each PLF hardware generation is organized in a different way and represents a distinct approach to the problem. Nonetheless, the basic computational pipeline that performs the matrix-vector multiplications of the PLF on DNA data (Figure 2) occupies approximately the same amount of resources as the respective pipelined 3rd generation data-path (Figure 3 in [13]). For this comparison, our 4th generation PLF pipeline was mapped to the same FPGA used in [13] (Virtex 5 SX95T-2). A significant difference between the two generations is that, the 4th generation pipeline is optimized for DNA data and can currently not accommodate statistical models for among site rate heterogeneity (see, e.g., [29] or [32]). The PLF pipeline has an initial latency of 115 clock cycles. Thereafter, it is able to compute one ancestral probability value during each clock cycle with a clock frequency of 293.3 MHz.

To evaluate the new PLF architecture, we compare it to the RAxML reference implementation on a state-of-the-art CPU (Intel i7 2600 at 3.4 GHz). Since this CPU provides 256-bit wide AVX vector instructions, we used the most efficient version of the PLF that deploys

AVX vector intrinsics (available in RAxML-Light: https://github.com/stamatak/RAxML-Light-1.0.5). To conduct a fair comparison between the CPU and the FPGA we introduce a new metric called VEUPS (Vector Entry Updates Per Second). A similar metric (CUPS; Cell Updates Per Second) is used in Bioinformatics to compare performance of pair-wise alignment kernels among various hardware platforms [30], [31]. The vector-entry size depends on the number of states in the model. Therefore, VEUPS-based performance comparisons can be misleading if they do not refer to vector entries of the same size. We measured the VEUPS number of RAxML-Light on DNA data without accommodating for rate heterogeneity. Using one core of the Intel i7 CPU and a real-world DNA alignment, the peak CPU performance is 78.83 Mega VEUPS. The respective performance of a single FPGA likelihood core instance amounts to 73.54 Mega VEUPS.

This shows that, the per-core VEUPS performance on a x86 architecture (when using AVX) is better than the respective per-PLF core performance on a FPGA. Nonetheless, modern FPGAs allow for instantiating up to 8 such likelihood-cores. However, such a dense design with several PLF hardware cores requires additional routing effort. Therefore, the maximum operating clock frequency decreases to 167.78 MHz. We estimate that the peak device-performance that can be achieved for the PLF from the instantiation of 8 likelihood cores on a Virtex 6 FPGA amounts to 335.57 Mega VEUPS (approximately 42 Mega VEUPS per instantiated core).

VII. Conclusion

We have presented the 4th generation of our series of architectures for computing the PLF on reconfigurable logic. The architecture is adapted to the computational requirements of modern tree search strategies. Approximately 60-70% of the core PLF calculations for ML-based programs and even a larger fraction for Bayesian programs that do not require explicit branch length optimization can thereby be offloaded to a FPGA. The new design is optimized for DNA data and can easily be adapted to support data with more states (e.g., protein data). We find that, for PLF calculations, current state-of-the-art FPGA devices achieve a speedup of factor four over single, high-end CPU cores.

In addition, we have presented (and emulated in SW) a novel approach for managing memory resources on the FPGA by maintaining a consistent view of the on-board FPGA memory organization on the host architecture that also steers computations. This approach can be deployed with real world applications such as RAxML to efficiently orchestrate the scarce and performance critical on-board memory resources.

With respect to future work, we plan to devise a method for transmitting data from the CPU to the external memory on the FPGA board via PCIExpress. Thus will allow us to design a fully functional system for exploiting the potential of our 4th generation reconfigurable architecture as accelerator for PLF calculations.

Acknowledgments

This work was partially funded under the auspices of the Emmy-Noether program of the German Science Foundation.

References

[1] S. Smith, J. Beaulieu, A. Stamatakis, and M. Donoghue, "Understanding angiosperm diversification using small and large phylogenetic trees," *American Journal of Botany*, vol. 98, no. 3, pp. 404–414, 2011.

[2] W. Fitch and E. Margoliash, "Construction of phylogenetic trees," *Science*, vol. 155, no. 3760, pp. 279–284, 1967.

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

[4] S. Roch, "A short proof that phylogenetic tree reconstruction by maximum likelihood is hard," *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, vol. 3, no. 1, pp. 92–94, 2006.

[5] 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.

[6] D. Zwickl, "Genetic Algorithm Approaches for the Phylogenetic Analysis of Large Biological Sequence Datasets under the Maximum Likelihood Criterion," Ph.D. dissertation, University of Texas at Austin, April 2006.

[7] S. Guindon and O. Gascuel, "A Simple, Fast, and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood." *Syst. Biol.*, vol. 52, no. 5, pp. 696–704, 2003.

[8] F. Ronquist and J. Huelsenbeck, "MrBayes 3: Bayesian phylogenetic inference under mixed models," *Bioinformatics*, vol. 19, no. 12, pp. 1572–1574, 2003.

[9] N. Lartillot, S. Blanquart, and T. Lepage, "PhyloBayes. v2. 3," 2007.

[10] M. Ott, J. Zola, S. Aluru, and A. Stamatakis, "Large-scale Maximum Likelihood-based Phylogenetic Analysis on the IBM BlueGene/L," in *Proc. of IEEE/ACM Supercomputing Conference 2007 (SC2007)*, 2007.

[11] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "Exploring FPGAs for accelerating the phylogenetic likelihood function," in *Proceedings of IPDPS 2009 (HICOMB)*, Rome, Italy, 2009, pp. 1–8.

[12] N. Alachiotis, A. Stamatakis, E. Sotiriades, and A. Dollas, "A reconfigurable architecture for the phylogenetic likelihood function," in *FPL 2009*. IEEE, 2009, pp. 674–678.

[13] N. Alachiotis and A. Stamatakis, "A generic and versatile architecture for inference of evolutionary trees under maximum likelihood," in *Proc. Conf Signals, Systems and Computers (ASILOMAR) Record of the Forty Fourth Asilomar Conf*, 2010, pp. 829–835.

[14] T. Mak and K. Lam, "Embedded computation of maximum-likelihood phylogeny inference using platform FPGA," in *Proc. of IEEE CSB 2004*, 2004, pp. 512–514.

[15] ——, "FPGA-Based Computation for Maximum Likelihood Phylogenetic Tree Evaluation," *Lecture Notes in Computer Science*, pp. 1076–1079, 2004.

[16] T. Jukes and C. Cantor, "Evolution of protein molecules," *Mammalian Protein Metabolism*, vol. 3, pp. 21–132, 1969.

[17] J. Ripplinger and J. Sullivan, "Does Choice in Model Selection Affect Maximum Likelihood Analysis?" *Syst. Biol.*, vol. 57, no. 1, pp. 76–85, 2008.

[18] J. Davis, S. Akella, and P. Waddell, "Accelerating phylogenetics computing on the desktop: experiments with executing UPGMA in programmable logic," in *Proceedings of 26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, vol. 2, 2004.

[19] J. Bakos, "FPGA Acceleration of Gene Rearrangement Analysis," in *Proceedings of 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 85–94.

[20] J. Bakos, P. Elenis, and J. Tang, "FPGA Acceleration of Phylogeny Reconstruction for Whole Genome Data," in *Proceedings of the 7th IEEE International Conference on Bioinformatics and Bioengineering, 2007*, 2007, pp. 888–895.

[21] B. Moret, J. Tang, L. Wang, and T. Warnow, "Steps toward accurate reconstructions of phylogenies from gene-order data," *Journal of Computer and System Sciences*, vol. 65, no. 3, pp. 508–525, 2002.

[22] S. Zierke and J. Bakos, "FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods," *BMC Bioinformatics*, vol. 11, no. 1, p. 184, 2010.

[23] A. Stamatakis, *Bioinformatics: High Performance Parallel Computer Architectures*. CRC Press, Taylor & Francis, 2010, ch. Orchestrating the Phylogenetic Likelihood Function on Emerging Parallel Architectures, pp. 85–115.

[24] S. Kasap and K. Benkrid, "High Performance Phylogenetic Analysis With Maximum Parsimony on Reconfigurable Hardware," *VLSI Systems, IEEE Trans. on*, no. 99, pp. 1–13, 2010.

[25] P. A. Goloboff, S. A. Catalano, J. M. Mirande, C. A. Szumik, J. S. Arias, M. Källersjö, and J. S. Farris, "Phylogenetic analysis of 73060 taxa corroborates major eukaryotic groups," *Cladistics*, vol. 25, pp. 1–20, 2009.

[26] D. L. Swofford, *PAUP*: Phylogenetic analysis using parsimony (* and other methods), version 4.0b10*. Sinauer Associates, 2002.

[27] N. Alachiotis and A. Stamatakis, "FPGA Acceleration of the Phylogenetic Parsimony Kernel?" *International Conference on Field Programmable Logic and Applications*, vol. 0, pp. 417–422, 2011.

[28] N. Alachiotis, S. A. Berger, and A. Stamatakis, "Efficient PC-FPGA Communication over Gigabit Ethernet," in *CIT*, 2010, pp. 1727–1734.

[29] A. Stamatakis, "Phylogenetic Models of Rate Heterogeneity: A High Performance Computing Perspective," in *Proc. of IPDPS2006*, ser. HICOMB Workshop, Proceedings on CD, Rhodos, Greece, April 2006.

[30] Y. Liu, B. Schmidt, and D. Maskell, "Cudasw++2.0: enhanced smith-waterman protein database search on cuda-enabled gpus based on simt and virtualized simd abstractions," *BMC Research Notes*, vol. 3, no. 1, p. 93, 2010. [Online]. Available: http://www.biomedcentral.com/1756-0500/3/93

[31] N. Alachiotis, S. A. Berger, and A. Stamatakis, "Accelerating Phylogeny-Aware Short DNA Read Alignment with FPGAs," in *FCCM*, 2011, pp. 226–233.

[32] Z. Yang, "Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites," *J. Mol. Evol.*, vol. 39, pp. 306–314, 1994.