FPGA Acceleration of the Phylogenetic Parsimony Kernel?

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

***Abstract*—The phylogenetic parsimony function is a popular, discrete criterion for reconstructing evolutionary trees based on molecular sequence data. Parsimony strives to find the phylogenetic tree that explains the evolutionary history of organisms by the least number of mutations. Because parsimony is a discrete function, it should fit well to FPGAs. We present a versatile FPGA implementation of the parsimony function and compare its performance to a highly optimized SSE3- and AVX-vectorized software implementation. We find that, because of a particular constellation in our lab, the speedups that can be achieved by using an FPGA, are substantially less impressive, than usually reported in papers on FPGA acceleration of bioinformatics kernels. We conclude that, a competitive spirit between SW and HW application developers can contribute toward obtaining more objective performance comparisons.**

***Keywords*-FPGA; SIMD; parsimony; performance analysis**

I. Introduction

The inference of evolutionary (phylogenetic) trees from molecular sequence data has many important applications in biological and medical research (e.g., [1]).

Input: The input for a phylogenetic analysis is a list of organism names and their associated DNA sequence data. Since DNA sequences for distinct organisms typically have different lengths, a so-called multiple sequence alignment (MSA) of the DNA sequences is computed prior to conducting a phylogenetic analysis using character-based methods (Maximum Parsimony [2] or Maximum Likelihood [3]). The goal of MSA is to determine which nucleotides of the organisms share a common evolutionary history. Because nucleotide insertions or deletions may have occured during the evolutionary history of the organisms, deletion events are denoted by inserting the gap symbol - into the sequences during the MSA process. After the MSA step, all n sequences have the same length m, that is, the MSA has m alignment columns (also called: characters, sites, positions).

Output: The output of a phylogenetic analysis is an *unrooted* binary tree topology. The present-day organisms under study (for which DNA data *can* be sequenced) are assigned to the leaves (tips) of such a tree, whereas the inner nodes represent extinct common ancestors.

Combinatorial Optimization: To reconstruct a phylogenetic tree from a MSA, criteria are required to assess how well a specific tree topology explains (fits) the underlying molecular sequence data. One may think of this as an abstract function $f()$ that scores alternative tree topologies for a given MSA. The goal of phylogenetic algorithms is to find the tree topology with the best score according to $f()$, that is, phylogenetic inference is a combinatorial optimization problem. The algorithmic problem in phylogenetics is characterized by the number of possible distinct unrooted binary tree topologies for n organisms which is given by: $\prod_{i=3}^{n}(2i-5)$. Finding the best tree for MSA-based criteria $f()$ such as Maximum Likelihood [4] or Maximum Parsimony [5] is NP-hard. Apart from developing efficient heuristic search strategies, the optimization of the scoring function $f()$, that is invoked millions of times during a heuristic tree search, and hence dominates execution times, represents an important research objective in phylogenetics.

Likelihood and parsimony are currently among the most popular methods for phylogenetic inference. Compared to the likelihood criterion, parsimony requires significantly less memory and computations to calculate $f()$ on a given tree topology which is important for analyzing very large datasets [1]. Here, we focus on the acceleration of the parsimony kernel via a pipelined reconfigurable architecture and by deploying 256-bit wide AVX vector instructions on a general purpose CPU. The constellation at our lab is rather atypical, since NA is a computer engineer and AS has a parallel computing background and 10 years of experience in developing and tuning phylogeny programs (e.g., the widely-used RAxML code; the three main papers have accumulated $\approx$ 2000 citations on Google Scholar; March 24, 2011). Hence, there is a permanent, vivid discussion whether FPGAs are suitable for accelerating phylogenetic kernels or not.

Thus, we compare our HW design with a highly optimized (at the bit level) open-source SW implementation that deploys 128-bit SSE3 vector instructions and the relatively recent 256-bit AVX vector instructions to accelerate the parsimony kernel. We find that, given a realistic parsimony kernel usage model, there is no significant difference in execution speeds between a high-end FPGA and an Intel i7 processor with AVX support. However, there are substantial differences in engineering effort: The design, implementation, and verification of the hardware architecture required almost a month, while porting the already existing SSE3 vectorization to AVX required only half a day. To allow for reproduction of all results in this paper the hardware description of the architecture is available as open-source code at: http://wwwkramer.in.tum.de/exelixis/FPGA_MaxPars.tar.bz2.

The remainder of this paper is organized as follows: in

Section II we address related work and in Section III we describe how to compute the parsimony score on a tree. In Section IV we outline the architecture and in Section V we present performance results. We conclude in Section VI.

II. Related Work

Few phylogenetic kernels have been mapped to hardware. Mak and Lam [6], [7], Alachiotis *et al.* [8], [9], [10], and Zierke and Bakos [11] map the floating point intensive likelihood function to FPGAs. Davis et al. [12] presented an implementation of the UPGMA method (Unweighted Pair Group Method with Arithmetic Mean) which is a simple tree reconstruction algorithm that is practically not used for phylogenetic analyses any more. In [13], Bakos *et al.* focused on tree reconstruction using gene order data, that is, the arrangement of corresponding genes in the genomes of different organisms is used to reconstruct trees.

Kasap and Benkrid [14], [15] recently presented, the—to the best of our knowledge—first reconfigurable architecture for the parsimony kernel and assessed performance on a FPGA supercomputer by exploiting fine-grain and coarse-grain parallelism. The implementation is limited to trees with a maximum of 12 organisms, which are very small by todays standards; the largest published parsimony-based tree has 73,060 taxa [1]. The authors use an exhaustive search algorithm to evaluate all possible trees with 12 organisms in parallel for finding the tree with the best parsimony score. An evaluation of all possible trees, even in parallel, is evidently not possible for parsimony-based analyses of larger trees because of the super-exponential increase in the possible number of trees. Parsimony-based programs for large datasets deploy heuristic search strategies (e.g., Subtree Pruning and ReGrafting (SPR) or Tree Bisection and Reconnection (TBR)). These search strategies (as implemented for instance, in TNT, `parsimonator` (our code), or PAUP*) do not require a de-novo computation of the parsimony score, based on a full post-order tree traversal as implemented in [14], [15]. Instead, they only require the update of a comparatively small fraction of ancestral parsimony vectors. Hence, a fundamentally different approach to implementing the parsimony function on a reconfigurable architecture for such commonly used heuristic search strategies is required.

Kasap and Benkrid 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 implementation in PAUP* [16] and not with respect to the fastest-known implementation of parsimony in the TNT program package used in [1]. Unfortunately, neither PAUP, nor TNT are open-source and therefore do not allow for an accurate performance analysis and comparison of the parsimony kernel. Therefore, we use our in-house code `parsimonator` (available at: http://wwwkramer.in.tum.de/exelixis/software.html), which implements a representative, yet simple, search strategy based on SPR moves. The parsimony kernel in `parsimonator` is highly optimized and the program can compute parsimony trees on DNA datasets with up to 116,408 organisms and ten genes.

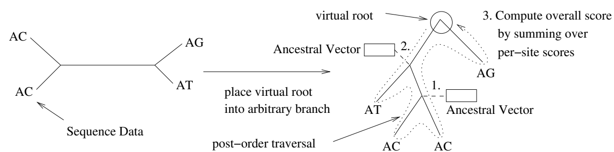

Figure 1. Virtual rooting and post-order traversal of a phylogenetic tree.

III. The Parsimony Kernel

The parsimony kernel operates directly on the MSA and the tree. The sequences in the MSA are assigned to the leaves of the tree and an overall score for the tree is computed via a post-order tree traversal with respect to a virtual root. An important property of the parsimony function is that parsimony scores are invariant to the placement of such a virtual root. Parsimony is characterized by two additional properties: (i) it assumes that MSA columns have evolved independently, that is, given a fixed tree topology, one can simultaneously compute the parsimony score for each MSA column in parallel. To obtain the overall score of the tree, the sum over all m per-column parsimony scores at the virtual root is computed. (ii) parsimony scores are computed via a post-order tree traversal that proceeds from the tips towards the virtual root and computes ancestral parsimony vectors of length m at each inner node that is visited (see Figure 1).

The parsimony criterion intends to minimize the number of nucleotide changes on a tree. Hence, for a given, fixed, tree topology we need to compute the smallest number of changes (mutations) required to generate the tree. This minimum number can be computed via a post-order traversal of the tree under consideration. Given an arbitrarily rooted tree, one can proceed bottom up from the tips toward the virtual root to compute ancestral parsimony vectors and count mutations, based on the two (previously computed; post-order traversal!) child vectors. To store tip vectors (containing the actual DNA data) and ancestral parsimony vectors (containing the ancestral sequences), we need to allocate 4 bits per alignment site m at each node of the tree. Hence, the total memory required to store the parsimony vectors is $m \cdot 4 \cdot (2n-2)$ bits, where m is the number of sites and $2n-2$ the total number of inner and outer nodes in an *unrooted* binary tree (n is the number of tips). In the following we will only describe the computational steps required to compute the parsimony score on a tree (please refer to [17] for a justification and further details).

The parsimony vectors (bit vectors) at the tips are initialized as follows: for a nucleotide `A` at a position i, where $i = 0...m-1$ we assign `A:=1000` (respectively `C:=0100`, `G:=0010`, `T:= 0001`). When the tip vectors

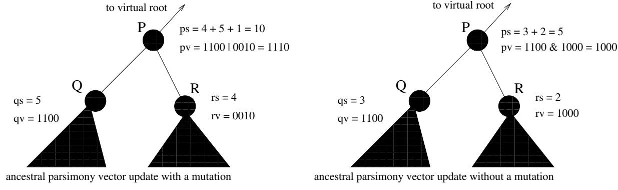

Figure 2. Parsimony vector and score updates with and without mutation.

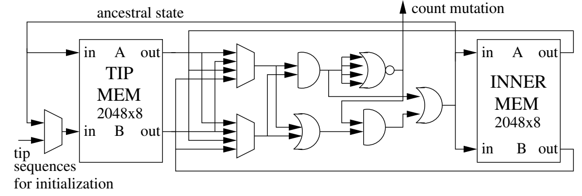

Figure 3. Architecture of the basic parsimony processing unit.

have been initialized, one can start computing the parsimony score of the tree. We will focus on computing the parsimony score ps_i (minimum number of mutations) for a single site i, since the overall score is simply the sum over all per-site scores at the virtual root: $\sum_{i=0}^{m-1} ps_i$.

Given two already computed child vectors q and r, we compute the parent vector p at site i as follows (see Figure 2). The parsimony score is initially set to the sum of the parsimony scores of two child vectors $ps_i := qs_i + rs_i$, that is, we take into account how many mutations were required to explain the two subtrees rooted at q and r for site i. Then, we compare the 4-bit vectors of q and r with a bit-wise `and` operation.

If this bit-wise `and` yields 0, this means, for instance, that site i in subtree q may only contain `A`s ($qv_i = 1000$) and r may only contain `C`s ($rv_i = 0100$) at site i. Hence, we need to add a mutation and increment the parsimony score by one $ps_i := ps_i + 1$. The parsimony vector at position i of p is then calculated as: $pv_i := qv_i\ OR_{bit-wise}\ rv_i$, that is, we conduct a bit-wise `or` on qv_i and rv_i to obtain a new state that now comprises `A` or `C`. Thus, $pv_i := 1100$, which means that the ancestral state can be `A` or `C` because we have already counted the required mutation. If the initial bit-wise `and` on qv_i and rv_i does not yield zero, we do not need to count a mutation, and simply set $pv_i := qv_i\ AND_{bit-wise}\ rv_i$, thereby essentially saving the shared state between qv_i and rv_i in pv_i. When the virtual root is reached, we conduct exactly the same computations on child vectors q and r for updating the parsimony score at the root p, but we do not require to store the ancestral state pv, since we are only interested in the score (mutation count) at p.

There exist only few open-source implementations of the parsimony kernel. The PHYLIP package [18] contains a proof-of-concept parsimony implementation that is not optimized at the bit-level. As already mentioned, we have recently released an optimized code called `parsimonator`. The `parsimonator` manual also includes a performance comparison of the non-vectorized, SSE3- and AVX-vectorized versions. We believe that this is currently the fastest open-source parsimony implementation with respect to the parsimony kernel implementation, albeit the search algorithm it uses is rather naïve because it is designed to generate starting trees for maximum likelihood analyses. The fastest available parsimony program is TNT [19]. PAUP* [16] is also a popular program for parsimony analysis, but significantly slower than TNT and `parsimonator`. Since we focus on designing an architecture for the parsimony kernel implementation, irrespective of the actual search algorithm, we use our in-house code to accurately measure and compare execution times.

IV. Reconfigurable Architecture

In the following we describe the reconfigurable parsimony architecture. We denote ancestral vector computations as *NV* operations and score computations at the virtual root by *EV*.

A. Processing Unit (PRU) Architecture

Figure 3 illustrates the basic processing unit (PRU) of the reconfigurable parsimony kernel. Each PRU operates on two child vector entries (two sites). The PRU architecture deploys a pair of dual-port memories, that is, one memory instance is used for storing tip vectors and one for storing inner vectors. Each memory instance can store a maximum of 2048 addressable bytes. The rationale for selecting this specific memory size is that, thereby we occupy a single 18Kb block RAM slice per PRU memory instance. If each PRU only requires a limited amount of memory blocks, the overall reconfigurable parsimony system can be extended by additional PRUs in a seamless way (see Subsection IV-B).

To initiate a parsimony analysis, only the TIP MEMORY has to be initialized with the bit-encoded DNA sequences in the MSA. Every tip and inner vector is assigned a static address space in the respective memory prior to executing any operation. During a post-order tree traversal, the following three memory access situations can occur regarding the input child vectors at nodes q and r: (i) q and r are both tips, (ii) either q is a tip or r is a tip, (ii) q and r are inner nodes. For the TIP-TIP *and* TIP-INNER cases, the input vectors are retrieved and read from the corresponding TIP and INNER memories, respectively. The result vector at node p (when it needs to be stored for a NV operation), is stored in the INNER memory. In analogy, a NV operation for the INNER-INNER case would require an INNER memory with three memory ports: two ports for reading the q and r vectors and a third port for writing the p vector. To efficiently implement the INNER-INNER case for NV (p, q, r are inner nodes) using present FPGA technology that only provides two ports per memory block, the p vector is temporarily stored in a special memory. We denote this special memory, which forms part of the TIP MEMORY, as EXTRA space. At each clock cycle, two 4to1 multiplexers (see Figure 3) are used to select the correct memory buses

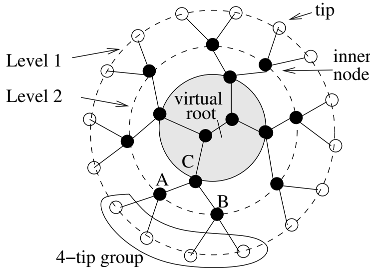

Figure 4. Worst-case tree topology in terms of EXTRA space requirements.

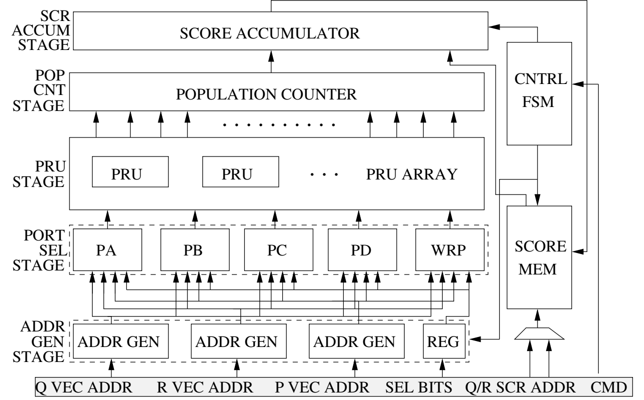

Figure 5. Top-level design of the pipelined architecture.

with valid vector input data. The multiplexer selection bits signal the corresponding TIP-TIP, TIP-INNER, and INNER-INNER cases. The group of logic gates in the center of Figure 3 implements the bit-wise operations to compute the parsimony kernel (see Section III). An additional 2to1 multiplexer is used to distinguish between the two data buses that provide input to the second TIP MEMORY port. One bus provides the DNA sequences of the MSA during the memory initialization process while the second bus provides ancestral states during NV and EV operations if the EXTRA space of the TIP MEMORY is used.

The size of the EXTRA space depends on the dimension of the input dataset, that is, the number n of taxa in the MSA and the number m of nucleotides per DNA sequence. It also depends on the tree shape. Figure 4 illustrates the worst-case tree in terms of EXTRA space requirements. Fully balanced trees require maximum EXTRA space, which amounts 50% of the memory required to store the input tip sequences in TIP MEMORY. In a fully balanced tree, for every group of 4 tips, one inner node needs to be stored in EXTRA space. In Figure 4, the highlighted group of 4 tips at Level 1 (tip level) has two parent nodes/vectors at Level 2 (one level closer to the virtual root). Vectors A and B (the direct ancestors of the tips), can be stored in INNER MEMORY. Since both, A and B, are inner vectors (stored in INNER MEMORY), their common ancestor C must be stored in EXTRA space to avoid a memory port conflict. In this worst case scenario, every inner vector in the highlighted grey area of Figure 4 must be stored in EXTRA space. Decisions for writing inner vectors to EXTRA space are orchestrated by an appropriately adapted `parsimonator` version. Thus, a dedicated, reconfigurable EXTRA space control unit is not necessary. For a fully balanced tree with n tips, the maximum number of inner nodes IN_EX that need to be stored in EXTRA space during a phylogenetic analysis is given by the following equation:

$$IN_EX = \begin{cases} n/2 - 2 & \text{if } n \mod 4 = 0 \\ (n + 4 - n \mod 4)/2 - 2 & \text{if } n \mod 4 \neq 0 \end{cases}$$

B. Pipelined Datapath

The generic input command that must be issued to the pipeline during a clock cycle to initiate parsimony computations is highlighted at the bottom of Figure 5. Both operations (NV and EV) require a set of four (two-byte long) read addresses; they contain the start addresses of the parsimony child vectors (*Q VEC ADDR*, *R VEC ADDR*) and corresponding parsimony scores (*Q SCR ADDR*, *R SCR ADDR*) of child nodes q and r. A *NV* operation also requires two additional write addresses for storing the parent vector p and the respective score at p.

The top-level design of the pipelined datapath has five stages (see Figure 5). Read/write memory addresses are initially generated by using 11-bit counters during the address generation stage (ADDR GEN). In the next pipeline stage (PORT SEL), the PA, PB, PC, and PD components implement multiplexers and logic that decides to which three (out of four) PRU memory ports the read/write addresses will be sent. The WRP component generates the write enable signal for the selected write port.

The PRU array comprises several parallel and completely independent PRUs. The number of PRUs determines the array size. All PRUs in the array receive the same read/write addresses, write enable signal, and selection bits to perform the same operation on different parsimony vector entries (sites of the MSA). Each PRU contains two memory components and logic. Since the PRU array is a vector-like component (each PRU operates independently), using only two 18Kb block ram slices allows for tailoring the array size to the available FPGA according to the number of unoccupied memory blocks on the device. Therefore, we implemented a program called *FPGA_MaxPars_Gen* (included in FPGA_MaxPars.tar.bz2) for generating VHDL wrapper files. Thereby, one can instantiate PRU arrays with user-defined length.

The POP CNT pipeline stage contains a population counter (to count the number of bits set to 1) for the computation of partial parsimony scores across alignment sites (PRUs). If one regards the tip and inner memory instances of the PRU array as two larger memory components, each with depth of 2048 and an array-size dependent memory line length, a partial score refers to the total number of mutations across sites that can be stored in a PRU array memory line (see Section IV-C for details on

the population counter).

Finally, in the SCR ACCUM stage, the parsimony score is computed with three 32-bit adders. The SCORE MEM memory is used to store intermediate scores (parsimony scores for each inner node that defines a subtree). One adder is used to calculate the sum of the input child node scores at q and r. The scores for q and r are retrieved from the SCORE MEM, based on the input SCR ADDR addresses. The second adder/accumulator sums up the partial scores that are produced by the population counter. The last adder computes the final score by adding the output of the accumulator to the sum of the scores at q and r. For *NV* operations, the parsimony score is written to SCORE MEM.

The size of SCORE MEM (2048 integers) is the upper limit for the number n of organisms (DNA sequences) our architecture can accommodate. Accordingly, this maximum number of 2048 organisms decreases proportionally with the number of PRU array memory lines required by each sequence. This number of PRU array memory lines increases with the number m of MSA sites/columns.

C. Population Counter

The population counter is implemented as a tree of adders with increasing width, that is, at each level of the adder-tree the adders are one bit wider than the previous level. The input bit-vector size for the first level depends on the PRU array size. The stand-alone FPGA_MaxPars_Gen software can be used to generate a population counter with a user-defined size and latency. Pipeline registers are optionally inserted as needed between adder levels in the tree to alleviate the negative impact of a very large (deep) population counter component on the overall operating clock frequency. To the best of our knowledge, FPGA_MaxPars_Gen is the only open-source population counter generator for FPGAs. Because the latency of the population counter influences the total latency of the parsimony architecture pipeline, FPGA_MaxPars_Gen instantiates a shift register (in the VHDL wrapper file) to synchronize the population count computations with the rest of the system.

V. Implementation, Verification, and Results

We describe the verification of the reconfigurable architecture in Section V-A. Then, we present a PC-FPGA prototype system (Section V-B) and a performance evaluation for a larger reconfigurable system (Section V-C).

A. Verification of the Parsimony Architecture

Initially, we modeled our architecture (including the address assignment to EXTRA space) in `parsimonator` using C. We replaced the standard *NV* and *EV* functions (accounting for 99% of total execution time) by implementations that reflected our reconfigurable architecture to assess and confirm the correctness of our approach.

Thereafter, the reconfigurable architecture was implemented in VHDL and mapped on a Virtex 5 SX95T-1 FPGA. We verified the correctness of the hardware system by extensive post place and route simulations using Modelsim 6.3f by Mentor Graphics and tests on an actual chip using a HTG-V5-PCIE development board with a Virtex 5 SX95T FPGA. Chipscope Pro Analyzer was used to monitor the input and output ports of the design.

Table I
Resources/Performance of Virtex 5 and Virtex 6 systems.

	64-PRU System	512-PRU System
Device	Virtex 5 SX95T	Virtex 6 SX475T
Slice Registers	5,568(9%)	41,091(6%)
Slice LUTs	4,133(7%)	22,520(7%)
Occupied Slices	1,933(13%)	9,608(12%)
Block Rams (18Kb)	132(27%)	1,028(48%)
Frequency (MHz)	192.374	188.456

B. PC-FPGA Prototype System

After successful verification, a fully operational PC-FPGA prototype system was designed to test this implementation of `parsimonator` using actual biological datasets on an actual board. We used the C interface of our open-source PC $\leftrightarrow$ FPGA communication platform [20] available at http://opencores.org/project,pc_fpga_com to transfer bit-encoded DNA sequences and issue 13-byte long NV/EV commands to the board. On the FPGA side, the DNA sequences were used to initialize the TIP MEMORY, and the NV/EV commands to trigger computations. The receiving background reader mechanism provided by this platform was used to receive parsimony scores on the PC side after an EV command had been issued to the board. The FPGA_MaxPars_Gen program was used to generate a PRU-array of size/width 64 as well as a correctly sized population counter for the prototype system, that is, 64 PRUs were placed in parallel allowing 128 alignment sites to be processed simultaneously (remember that each PRU can compute the parsimony score for two sites; see Section IV-A).

C. Results

To present a fair performance assessment for our accelerator architecture we created a high performance instance of our architecture (using FPGA_MaxPars_Gen) with an array of 512 PRUs and mapped it on a Virtex 6 SX475T-2 FPGA. Furthermore, we vectorized `parsimonator` with 256-bit wide AVX SIMD instructions. An evaluation of the prototype and high performance systems regarding resources and clock frequencies is provided in Table I. Note that, the currently largest available FPGA with respect to available block RAM slices (Virtex 7 VX865T) can accommodate an array of 1800 PRUs and thus allows for computing 3600 sites in parallel.

Table II shows execution times (in seconds) for real-world biological datasets using the SSE3 and AVX versions of `parsimonator` (using one core of an Intel i7-2600 CPU at 3.40GHz) and the reconfigurable architecture with 512 PRUs (mapped on the Virtex 6 device). The FPGA accelerator is up to 9.65 times faster than the optimized software.

Table II
EXECUTION TIMES (IN SECONDS) FOR NV/EV INVOCATIONS.

Dataset	Execution Times			Speedup FPGA vs	
#taxa-#sites	SSE3	AVX	FPGA	SSE3	AVX
100-48965	1.48	0.91	0.15	10.12	6.22
125-16503	1.59	0.92	0.16	9.69	5.6
150-871	0.13	0.10	0.01	12.55	9.65
218-1318	0.40	0.26	0.04	9.08	5.9
354-224	0.24	0.21	0.04	6.41	5.61
500-759	0.74	0.52	0.06	11.56	8.12

VI. CONCLUSION

We have presented a reconfigurable architecture for computing the parsimony function on evolutionary trees of realistic size. The architecture is adapted to the computational requirements of modern parsimony search strategies and has been demonstrated to work in a full PC-FPGA setup, where the PC steers the computations on the board by using a PC $\leftrightarrow$ FPGA communication library. We also demonstrate how, sometimes overarching expectations with respect to speedups by FPGA-accelerated discrete functions, can be alleviated by a close collaboration and competition between SW and HW engineers and application domain specialists. Using one of the fastest currently available CPUs with 256-bit AVX instructions, we show that, if the reference SW is properly optimized and the capabilities of modern CPUs are rigorously exploited, FPGAs are slower than expected or hoped for.

ACKNOWLEDGEMENTS

This paper is dedicated to the memory of Walter M. Fitch. He was one of the pioneers of computational phylogenetics and parsimony methods.

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

REFERENCES

[1] 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.

[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," *IEEE/ACM Trans. on Comp. Biology and Bioinformatics*, pp. 92–94, 2006.

[5] W. Day, D. Johnson, and D. Sankoff, "The computational complexity of inferring rooted phylogenies by parsimony," *Mathematical biosciences*, vol. 81, no. 33-42, p. 299, 1986.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] S. Kasap and K. Benkrid, "A high performance FPGA-based core for phylogenetic analysis with Maximum Parsimony method," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009, pp. 271–277.

[15] ——, "High Performance Phylogenetic Analysis With Maximum Parsimony on Reconfigurable Hardware," *VLSI Systems, IEEE Trans. on*, no. 99, pp. 1–13, 2010.

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

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

[18] J. Felsenstein, "Phylip (phylogeny inference package) version 3.6," 2004, distributed by the author. Department of Genome Sciences, University of Washington, Seattle.

[19] P. Goloboff, "Analyzing large data sets in reasonable times: solution for composite optima," *Cladistics*, vol. 15, pp. 415–428, 1999.

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