Accelerating the phylogenetic likelihood function using domain-specific processors and an FPGA-based scaling unit for large-scale phylogenies

GEERT ROKS, University of Twente, The Netherlands

Phylogenetics study the evolutionary history of organisms using an iterative procedure of creating and evaluating phylogenetic trees. This procedure is highly compute-intensive; constructing a large phylogenetic tree requires hundreds to thousands of CPU hours. Most phylogenetic analyses today rely either on maximum likelihood (ML) or Bayesian inference (BI) methods for inferring phylogenetic trees; the phylogenetic likelihood function (PLF) is employed in both ML and BI approaches as the tree-evaluation function, accounting for up to 95% of the overall analysis time. This work explorers the AMD Versal™ Adaptive SoC for acceleration of the PLF. This novel architecture provides a tight integration of domain-specific processors (AI Engines) with programmable logic (PL), which seems highly suitable for calculation of the PLF. The core of the PLF calculation (matrix multiplication) is mapped to the AI Engines, while custom logic on the PL handles data movement and data organization, as well as numerical scaling which is a prerequisite for yielding numerically stable solutions for large-scale phylogenetic studies. A thorough design-space exploration identifies bottlenecks and platform limitations, to inform the best mapping of the PLF to the architecture. The final system is up to 3.2× faster than the current fastest FPGA-based implementation, where data movement optimizations could improve this up to 11×. Results also show between 23.8× and 47.0× higher computational power of the Versal SoC than one modern x86 CPU (both AMD and Intel) using AVX2 intrinsics and even up to 1.9× higher performance than 128 high-end CPU cores. The exploration provided by this work could be used as a guide for future acceleration efforts that benefit from close integration between an array of vector processors and programmable logic.

Additional Key Words and Phrases: Phylogenetic Likelihood Function, RAxML, Versal™Adaptive SoC, Heterogeneous computing, AI engine, Programmable Logic

1 INTRODUCTION

Phylogenetics is the study of the evolutionary history of organisms and mainly concerns itself with understanding the relationships between organisms in the context of evolution. A common tool in phylogenetic research is the phylogenetic analysis using the DNA, RNA or protein sequences of the organisms of interest. The evolutionary relationships are derived from the differences and commonalities of these sequences. The understanding of these relationships leads to many scientific insights in nearly every branch of biology, but also outside of biology. Epidemiologists use phylogenetic analysis to reconstruct the spread of viruses within and between countries, for example during the COVID-19 pandemic[1, 33, 55], while in paleoanthropology it is used to understand phylogenetic relationships between fossils [41]. Phylogenetic analysis is also applied in research into algae for production of biomass [20] and in therapy development for patients with bowel diseases [30]. It is also getting applied more and more commonly in linguistics for research into language evolution [22].

The result of a phylogenetic analysis is called a phylogeny usually represented as a phylogenetic tree. This is a binary tree diagram that represents the relationships between the provided sequences, referred to as taxa. The taxa are at the leaves/tip nodes of the tree and the internal nodes represent extinct common ancestors of the taxa. The length of the edges, also referred to as branches, represent the evolutionary time that has passed between the species at the nodes that the edge connects. The tree structure has no root

Author's address: Geert Roks, University of Twente, Drienerlolaan 5, Enschede, The Netherlands, 7522 NB.

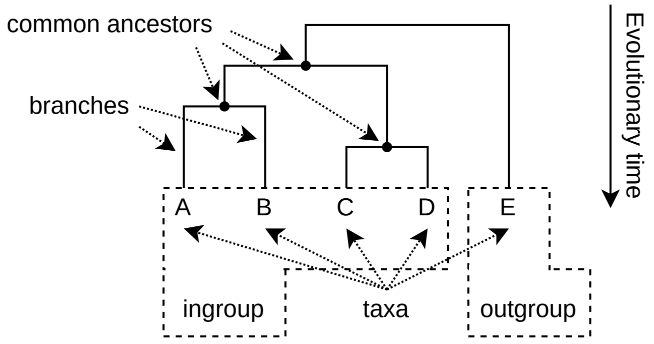


Fig. 1. Generic phylogenetic tree with terminology

node, but by adding an outgroup to the analysis, the structure of the ingroup becomes rooted. The ingroup is the group of taxa that are of interest and the outgroup is a taxon that is known to be diverged from the ingroup before the ingroup diverged from each other. Figure 1 shows a generic phylogenetic tree with the terminology.

There are multiple methods of inferring the phylogenetic tree from the genomic data of the taxa. The two main approaches are distance- and character-based. The distance-based approaches compare the distance between of each pair of sequences and constructs a tree from this distance matrix. However character-based methods are generally preferred due to the fact that distance-based methods tend to perform poorly with distantly related species[26, 61]. The character-based approach consists of the Maximum Parismony (MP)[26], Maximum Likelihood (ML)[16] and Bayesian Inference (BI)[42] methods. These methods compare every sequence at the same time, calculating a score for each character to combine into a score for the complete tree. The MP method calculates the score of the tree on the least amount of nucleotide or amino-acid changes that explains the observed data. Whereas the ML and BI methods use a statistical model and the likelihood value to score the tree. The advantage of MP is its simplicity and needs relatively little computation power as compared to the likelihood-based methods. However, since MP does not use a model, it is unable to incorporate any prior knowledge about the process of sequence evolution, like multiple substitutions over a branch[64]. This makes it very susceptible to the bias of Long-Branch Attraction (LBA)[15], which is the classifying of long branches with many nucleotide changes as closer related than they are in reality. Likelihood-based methods can also suffer from this bias if the model is incorrect or too simplistic[51]. However, the model-based approach of ML and BI also allows for updating to a more realistic model if necessary. Therefore, the likelihood-based methods are considered to be more generally applicable, especially for larger datasets[26, 63, 64, 73]. To find the tree with the highest score, theoretically every tree should be calculated, but this is not feasible, especially with large phylogenies. Cavalli-Sforza et al. [9] show that the number of possible unrooted trees with n taxa is $\frac{(2n-5)!}{(n-3)!2^{n-3}}$, meaning that a tree with 100 taxa has $1.7 \cdot 10^{182}$ possible trees that need to be scored. Therefore, heuristic tree searching approaches, like Subtree Prune and Regraft (SPR) moves[53], are often used. Although, this does not guarantee that the tree with the highest score will be found, it does make it feasible to find a good tree candidate within a reasonable amount of time. Recently there are also efforts using machine learning to perform phylogenetic inference[35, 48, 52, 72], which show promise to address fundamental challenges in the technology and aims to provide a general model that takes any input data to reduce computation during inference compared to the methods mentioned above. However, these methods are still limited to small trees and generally considered to not be of the same quality as the traditional inference methods [34], especially quality training data is holding this technique back.

Since the introduction of next-generation sequencing techniques[24], it has been easier to get a lot of accurate DNA, RNA or protein data. So, more studies are using large datasets in their phylogenetic research[27, 37, 38, 49], with datasets getting as large as 1367 taxa each with > 3 million sites [31]. However, when dealing with a large dataset, either many-taxa or large genomes, it means that the scoring function needs to be executed very often. ML is a broadly accepted method to score phylogenetic trees[40], which uses the Phylogenetic Likelihood Function (PLF) [16] as a scoring function. Widely used likelihood-based phylogenetic tools, such as RAxML [50] and MrBayes[45], spent approximately 85% to 95% of their execution time on the PLF[2]. Acceleration of the PLF is needed to construct reliable large-scale phylogenies within an acceptable time frame. There have already been efforts to explore the acceleration of the PLF using CPUs[17, 39], GPUs[43, 46] and Field-Programmable Gate Arrays (FPGAs)[3, 32, 71]. Also, there are efforts that investigate algorithmic solutions to reduce analysis time[11, 21, 36].

The calculation of the PLF relies on many back-to-back matrix multiplications with values on the unit interval ($[0, 1]$). Many multiplications of these values, make them very small and results in underflow errors in the implementation. Therefore, double-precision floating-point (DP) arithmetic is often used to reduce this underflow error. However, this is computationally and area expensive. Additionally, tools like RAxML[50] scale values that falls below a certain threshold by multiplying them with a large constant, to keep the values from underflowing. Berger and Stamatakis [8] show that the use of single-precision floating-point (SP) arithmetic provides sufficient accuracy for tree searches up to 2000 taxa. Their results show a reduction of 40% in PLF execution time due to increased cache efficiency and a 50% reduction in memory footprint due to it only using half of the amount of bits compared to DP. However, the smallest value that the normal range of a SP can represent is 2^{-126}, whereas a DP can represent a value as low as 2^{-1023}. This will result in much more frequent scaling operations during runtime which can greatly diminish the performance gain from using SP [8].

Due to the increased interest in Artificial Intelligence (AI) recently, there are some developments in domain specific accelerators, most notably the AMD Versal™ Adaptive System-on-Chip (SoC)[18]. These SoCs contain multi-core general processors, Programmable-Logic (PL) and an array of domain specific processors for the acceleration of AI workloads and digital signal processing, referred to as AI Engine (AIE). The core of AI calculations is based on matrix multiplications, which is also the underlying operation for the PLF and therefore this work investigates the AMD Versal™ SoC to accelerator the PLF. However, since it is conventional to use SP arithmetic in the AI domain, because there is no need for more precision. In fact there are many efforts trying to reduce the number of bits using quantization[10, 28, 29]. Although this limitation confines any PLF implementation on the Versal™ SoC to SP, its architecture is particularly well-suited for this acceleration task, due to close integration between the AIEs and PL. The underlying idea of our approach is to exploit the AIE array for SP matrix multiplication, which will get a constant stream of data to execution the computation from the PL that contains special-purpose data preparation and scaling units. Since data movement is the bottleneck for PLF acceleration [2, 3, 57], we expect that the combination of these two elements on the same SoC will result in a high throughput solution, that better supports large-scale phylogenetic analyses.

This thesis investigates the use of this new type of heterogeneous computing platform for accelerating likelihood-based phylogenetic tree reconstruction algorithms for large-scale phylogenetic analysis. The main research question is as follows: "Is the new type of heterogeneous computing platform with Programmable-Logic and array of AI processors on the same chip able to accelerate the Phylogenetic Likelihood Function to increase throughput compared to current software implementations and existing architectures in the literature?". To answer this research question the following sub-questions will need to be answered:

- Which mapping of the PLF to the AIE and PL on a Versal™ Adaptive SoC results in the shortest execution time?
- How close is the fastest implementation to the limits of the architecture and what resources poses to be the highest bottleneck?
- How does the fastest implementation on the new platform compare to the state-of-the-art CPU implementation and previous acceleration efforts?

The contributions that this thesis provides to the existing body of work is as follows:

- This is the first, to the best of the authors' knowledge, exploration of the novel Versal™ Adaptive SoCs for the acceleration of likelihood-based phylogenetic analyses. The tight integration of dedicated matrix multiplication accelerators with programmable logic results in a highly suitable platform for scientific workloads. This is demonstrated by boosting performance of the Phylogenetic Likelihood Function (PLF), which is ubiquitous in phylogenetics.
- An exploration of the design space of the platform is provided to find a high-performance mapping for the data and computation needs of the PLF. Additionally, this work may provide a useful guide for other acceleration efforts that benefit from close PL-AIE integration by identifying platform limitations and bottlenecks. The resulting acceleration system is up to 3.2× and 3.0× faster than other special-purpose FPGA architectures [32] and a single x86 CPU core using AVX2, respectively, with performance primarily limited by data movement over PCIe. Data movement optimizations explored by other works could improve this to 11× and 10.6× respectively.

The rest of this thesis will first discuss the background (Section 2) and previous literature (Section 3), followed by a detailed description of the system architecture (Section 4) and its implementation onto a Versal™ Adaptive SoC (Section 5). Afterwards the system performance is evaluated (Section 6) and discussed with a look at future work (Section 7).

2 BACKGROUND

2.1 Phylogenetic Tree Reconstruction

There are multiple steps involved in reconstructing a phylogenetic tree from a set of DNA or protein data. Figure 2 shows a general top-level overview of the steps involved.

First off, biological material is sequenced to obtain the raw data. Then a set of data is selected for a particular analysis and the data is aligned using Multiple Sequence Alignment (MSA)[67]. In this work, the data is expected to be already sequenced and aligned. From the aligned data an initial tree topology

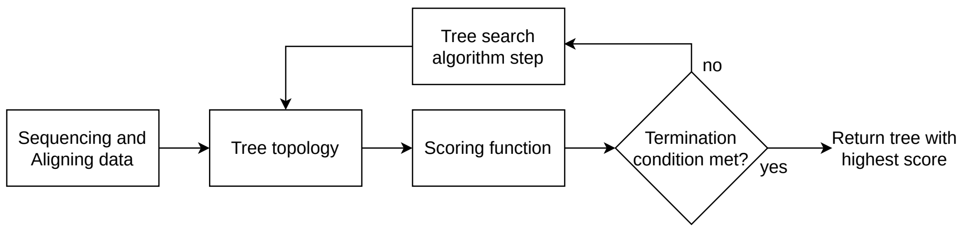

Tree search algorithm step
Sequencing and Aligning data
Tree topology
Scoring function
Termination condition met?
no
yes
Return tree with highest score

Fig. 2. A top-level overview of a phylogenetic reconstruction algorithm

is constructed often using a parsimony technique. This tree is then evaluated by a scoring function. Both ML and BI employ the PLF as the scoring function. Thereafter, a searching algorithm generates a new tree topology, which is also evaluated by the same scoring function. The algorithm is iterative and repeats these steps until certain termination conditions are met, for example the tree score did not improve in several iterations. When the searching algorithm terminates it will return the tree with the highest score.

As previously shown in Figure 1, a phylogenetic tree is a binary tree structure. The leaf nodes of the tree represent the biological entities in the analysis and the inner nodes are their extinct ancestors. These nodes are connected by branches with a particular length (t_i), which indicates evolutionary time between the two nodes. The node itself contains a sequence of data with certain amount of columns, also called alignment sites. Each site can have one of 4 states for DNA (A, C, G or T) and one of 20 states for protein data. In the rest of this work we assume to work with DNA data.

When analyzing a large tree, a lot of scores need to be calculated, which will take a very long time to do for every possible tree topology. For a 100 taxa tree there are $1.7 \cdot 10^{182}$ trees topologies, and even if it was possible to score a tree topology within one clock cycle, then it will still take $5.4 \cdot 10^{164}$ years on a hypothetical 10 Gigahertz CPU. Therefore, heuristics are employed to reduce the number of trees to be evaluated and allow phylogenetic studies to finish in a reasonable amount of time. However, this is at the cost that no guarantee can be given that the actual highest scoring tree is found.

A widely used heuristic searching algorithm is Subtree Prune and Regraft (SPR)[53]. Figure 3 shows a so-called SPR move, which consists of two steps: 1) removing a sub-tree from the main tree (pruning)

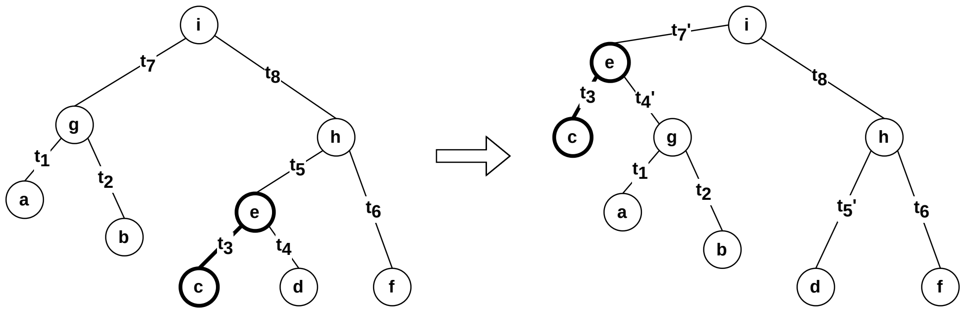
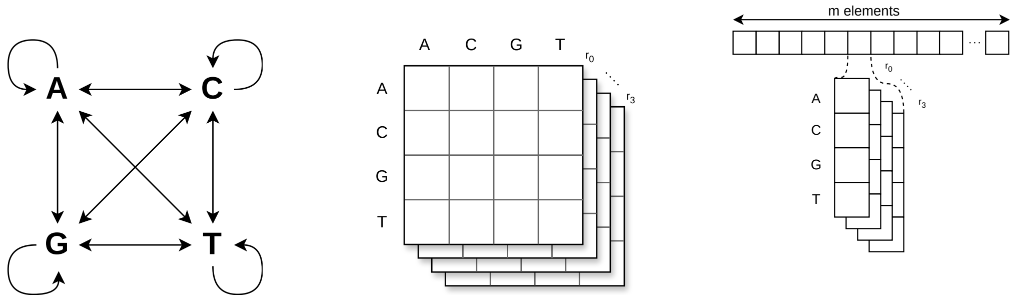


Fig. 3. An example of an SPR move on the sub-tree $\{e, c\}$. The subtree is pruned from branches $t5$ and $t4$ and regrafted on branch $t7$. The new branches that are created after the SPR move are indicated by the accent in their name

and 2) reattaching this sub-tree in a different location in the main tree (regrafting). The sub-tree that is removed is a node with one of its child sub-trees. The remaining child is then connected directly to its former grandparent. The removed sub-tree is then regrafted by splitting a branch somewhere else in the topology and inserting itself in between. After every SPR move, the likelihood of the tree is calculated. There are of course a large amount of possible SPR moves and choosing which move to execute is done by an SPR algorithm[23]. RAxML uses a technique called lazy subtree rearrangement, also called lazy SPR moves, where not all SPR moves for a particular tree are exhaustively tested, but the SPR moves are always performed on the tree that has the highest likelihood.

2.2 Phylogenetic Likelihood Function

Calculating the PLF for a given tree topology requires a fixed set of branch lengths and parameters for the nucleotide substitution model. In the case of DNA sequence data, the 4x4 matrix Q encapsulates the instantaneous probabilities of transitioning from one DNA nucleotide (A, C, G, or T) to another (A, C, G, or T), according to the substitution model. Most phylogenetic software tools implement the discrete Γ model of rate heterogeneity[60], which extends the basic model with additional parameters and introduces four discrete rates $\rho_0, \rho_1, \rho_2, \rho_3$. The Γ model provides a good approximation of the way that the substitution rates may vary per site. Gaut et al.[19] and Yang[62] have shown that models that allow for different rates at sites are more robust, than models assuming a fixed substitution rate for every site. Another common assumption is that the model is time reversible, which is know as the General Time-Reversible (GTR) model[2, 44]. A combination of these two models is often used in order to estimate the nucleotide substitution probabilities for the PLF. This is calculated for each rate category ρ_i given a branch of length t as $P(t, \rho_i) = e^{\rho_i Q t}$. Since Q is a matrix, the resulting branch matrix, referred to as substitution probability matrix (SPM), has the same dimensions (4×4 for DNA).



(a) Nucleotide substitution model (b) The branch substitution matrices under the Γ-model (c) Conditional likelihood vector under the Γ-model

Fig. 4. Diagrams of the phylogenetic model and data structures, adapted from [40]

Once the model parameters and branch lengths have been estimated, the conditional likelihood vectors (CLVs) at the inner nodes of the tree can be computed. Each CLV has the same length as the MSA input alignments of m alignment sites. Each site c, where $1 \leq c \leq m$, the CLV $\vec{L}^{(k)}(c)$ at inner node k contains the probabilities of observing nucleotides A, C, G or T at that site: $\vec{L}^{(k)}(c) = \left[L_A^{(k)}(c), L_C^{(k)}(c), L_G^{(k)}(c), L_T^{(k)}(c)\right]$.

For the Γ model, each site has four vectors, one for each discrete rate and each of these rate vectors have the four nucleotide probabilities, resulting in 16 floating-point numbers per site. So, although the CLVs are referred to as vectors they are in fact $(r \cdot s) \times m$ matrices, where r is the number of discrete rates and s the number of states. The likelihood vectors of the leaf nodes are constructed using the nucleotide data of the MSA input. A probability of 1.0 is noted for the nucleotide that is observed at that site in the input and the rest have a probability of 0.0.

The order in which CLVs at the inner nodes are calculated is dictated by the Felsenstein pruning algorithm[16]. The algorithm will recursively collapse (prune) a pair of child nodes l and r to update the CLV of their common ancestor parent node p. This starts from the tips and moves in a post-order tree traversal towards the virtual root vr. This virtual root is part of the Pulley Principle [16] and explained in section 2.4. Since the Γ-model, like many other popular models, allows for the substitution rates to vary per site, it also allows for the computation of the parental CLV to be done in a site by site manner. So, for a parent node p, its child nodes l and r, their fixed branch lengths t_l and t_r, and discrete rate ρ, the CLV entry for state X at site c in node p is computed as:

$$L_X^{(p)}(c) = \left(\sum_{S \in N} P_{XS}(t_l, \rho) L_S^{(l)}(c) \right) \left(\sum_{S \in N} P_{XS}(t_r, \rho) L_S^{(r)}(c) \right) \tag{1}$$

where N is the set of DNA states ($N = \{A, C, G, T\}$) and $X, S \in N$.

When the virtual root is reached, the CLV of the virtual root $\vec{L}^{(vr)}$ is transformed into a vector of per-site likelihoods l, which is also computed site by site. The following shows the per-site likelihood calculation $l(c)$ at site c:

$$l(c) = \sum_{S \in N} \pi_S L_S^{(vr)}(c) \tag{2}$$

The base frequencies $\pi_A, \pi_C, \pi_G, \pi_T$ are the a priori probabilities of observing an A, C, G or T nucleotide respectively. These are commonly obtained from the nucleotide frequencies in the input MSA, or all set to 0.25. The overall likelihood score of the tree is computed from this likelihood vector as the product of all the per-site likelihood scores of the m sites:

$$L = \Pi_{c=1}^{m} l(c) \tag{3}$$

RAxML implements the PLF calculation (Equation 1) as a sequence of matrix multiplications, as illustrated in Figure 5 for one alignment site c and one discrete rate ρ. The left and right child CLV entries are multiplied by the left and right SPMs respectively. The result of these two vector-matrix multiplications is then combined into a single vector by an element-wise multiplication, after which another vector-matrix multiplication with a matrix containing eigenvalues, referred to as eigenmatrix, results in the parent CLV entry for that site and rate. RAxML relies on eigenvalue decomposition for efficiently computing CLVs. This calculation is repeated four times per site (once for every Γ rate) with a dedicated SPM for each rate, which is also repeated for every site. The dedicated SPMs per rate stay the same for every site and the same eigenmatrix is used for all rates and sites.

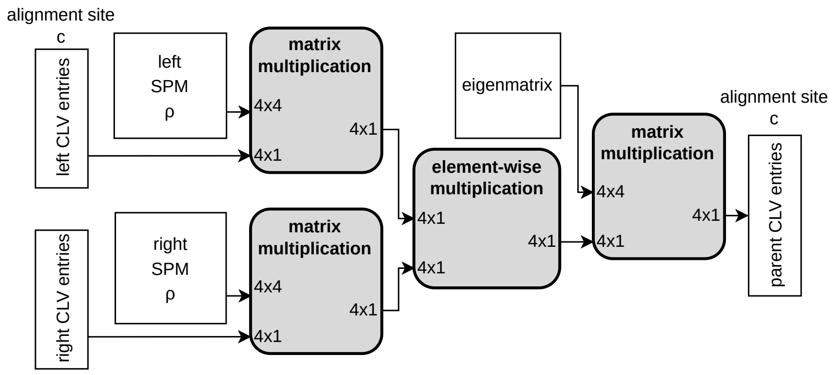


Fig. 5. Schematic overview of PLF implementation in RAxML for alignment site c and discrete rate ρ for DNA data. This calculation is repeated for every rate (four times under the Γ model) and every site of the input CLVs

2.3 Numerical stability for large phylogenies

Since all CLVs and SPMs contain probabilities (ie. values ≤ 1), the values can get extremely small when approaching the virtual root in large trees. So, to reduce numerical underflow in computing systems, the sum over the logarithm of the per-site likelihood scores is computed (instead of Equation 3).

$$\log L = \sum_{c=1}^{m} \log(l(c)) \tag{4}$$

Another measure to avoid numerical underflow and ensure numerical stability for large phylogenetic trees is scaling. If all probabilities of a certain site computed by Equation 1 fall below a threshold ϵ, then they are all scaled by multiplying each of them by a constant E. For RAxML[8, 50], the scaling constant E is chosen to at $E = 2^{256}$ for DP and for SP it is at $E = 2^{32}$, with the threshold set at $\epsilon = \frac{1}{E}$. To reverse scaling at the virtual root to calculate the log-likelihood score correctly, the total number of scaling events are counted for each site during a pruning step. These values are stored in the m-element vector $\vec{U}$. This vector is passed along towards the virtual root and at a pruning step, the vectors of both children are summed element-wise. When the virtual root is reached, the $\vec{U}$ vector contains a record of how many times each site has been scaled. Each per-site likelihood score at the virtual root is then scaled down by dividing by E for the amount of scaling events that took place for that site. Equation 2 is adapted as follows to account for scaling:

$$l(c) = \left(\frac{1}{E}\right)^{\vec{U}^{(vr)}(c)} \sum_{S \in N} \pi_S L_S^{(vr)}(c) \tag{5}$$

RAxML implements this method of scaling, but also implements a second method called fast scaling. For this method the input data is filtered at the start of an phylogenetic analysis to only include unique per site patterns, where a pattern is the specific combination of nucleotides for one site over all taxa in the input data. Vector $\vec{w}$ keeps track of how often a certain pattern occurs in the original input dataset. When RAxML is using the fast scaling method, it accumulates the occurrence number of the site that is scaled $\vec{w}(c)$ in register s. After a single PLF, s will hold the number of scaling events that happened for that step, but only has computed it for the number of unique patterns, because if duplicates of a pattern exist in the input data, then those would need the same computation as the first. When the virtual root is reached, s holds the total amount of scaling events for this tree evaluation and to reverse scaling, the overall likelihood score L is divided by $s \times E$. This method reduces both the memory utilization and computation time.

2.4 Branch length optimization

The pruning algorithm only works on fixed branch lengths, but the branch lengths for the initial tree topology are chosen at random. So these branch lengths need to be optimized in order to find the maximum likelihood score of the particular tree topology. The Pulley Principle, introduced by Felsenstein [16], allows for optimization of each branch individually. It states that the likelihood of an unrooted tree stays the same when a new node is placed on a branch, creating a rooted tree, as long as the sum of the branch lengths of the two children to their parent is constant. Resembling a pulley with a rope over it, where if the rope on one side of the pulley is pulled, this side will extend with the same amount as the other will shorten. It is important to note that the Pulley Principle relies on two assumptions. First, there should be no constraints on the branch lengths and second, the substitution rates should follow a reversible Markov process, like the GTR-model. From this principle we find that the rooted tree is equivalent to its unrooted version and even that a virtual root vr can be placed on any branch of the tree and the likelihood of the tree is not affected. Using this principle, a branch t between nodes n_1 and n_2 can be selected for optimization and the virtual root vr placed on this branch, creating branches t_1 and t_2 between the nodes and the vr. By placing vr right next to for instance n_1 in that $t_1 = 0$, it leads to $t = t_2$. If all branch lengths except the branch t with the vr on it is considered fixed, then a univariate optimization algorithm, such as the Newton-Raphson method [66], can be used to optimize t using only t_2. The likelihood score of vr is repeatedly calculated with only changing t_2 in order to find the branch length t that maximized the likelihood score. The procedure is than repeated for every branch in the tree.

2.5 Computing architecture

The AMD Versal™ Adaptive SoC is a novel computing chip architecture, containing programmable logic (PL) and an array of domain specific processors on the same SoC. This SoC is primarily focused on artificial intelligence (AI) and digital signal processing workloads. Therefore its domain specific processors are referred to as AI Engines (AIEs), which are very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units. This allows for three forms of parallelism: instruction-level parallelism due the VLIW allowing for up to 7 instructions per clock cycle, data-level parallelism due to vector instructions operating on multiple sets of data and multi-core parallelism, because the AIEs are organized in an array of independent processors. The VLIW processor includes up to two move operations, two vector loads, one vector store, one vector operation and one scalar operation.

Figure 6 shows a three layer schematic overview of the components of the AIE array. The left-hand side shows the AIE containing the memory/stream interfaces, scalar unit, vector unit, two load units and one store unit. The AIE processor is located on an AIE tile (middle part in figure), where the processor has connections to the neighboring tiles in the array and a 32 KiB memory module. A single AIE tile can access up to 3 neighboring (north, south and either east or west) memories directly, giving each tile direct memory access to a maximum of 128 KiB per AIE (local memory module + 3 neighbor memory modules), from

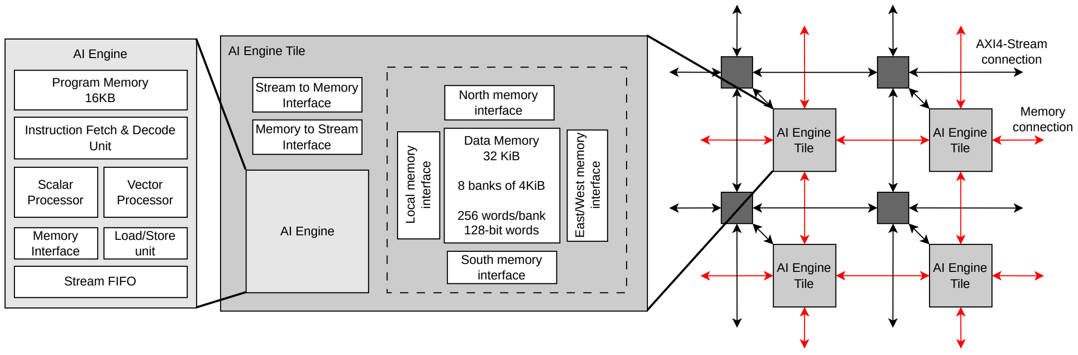


Fig. 6. Architectures of the AI Engine (AIE), the AIE tile and the AIE array (Adapted from [59])

which the AIE can read 2× 256-bit words and write 1× 256-bit words in a clock cycle . The AIE tile also has a stream interface that provides 2× 32-bit-word reads and 2× 32-bit word writes per clock cycle. These tiles are organized in an array (right part of the figure) interconnected with 32-bit streams that connect to the AXI4-Stream switches located next to every tile, which allows every AIE tile to communicate with any tile in the array via these streams. Additionally there is a specialized 2-deep 384-bit-wide cascade interface that connects neighboring tiles on one row in a single direction. The direction of the cascade stream alternates per row, for example on one row all of the cascade streams go to the east tile while on the row above the cascade streams go to the west tile. However, the cascade streams can only transport specific accumulator types and not regular floating-point types.

The AIE programming API is used to create AIE kernels, which are the program code of the AIE. Multiple specialized kernels can be designed for a solution which are organized in an AIE graph. The graph specifies the connection between kernels and optionally the location of the kernel. There are two mechanisms to transfer data between kernels: *windows*[1] and *streams*. A window transfers a fixed-size data chunk, whereas a stream operates as a FIFO (First In, First Out) buffer. Kernels that use windows must restart to retrieve the next window, which causes control overhead for starting and stopping kernels or maintaining local variables across consecutive kernel executions. Stream-based kernels can transfer data continuously as long as there is space in the buffer, with an added benefit that since they don't have to restart for new data, they can hold local variables for a long time.

AIE kernels can share a single AIE tile, or get pinned to a dedicated title. Kernels that share a tile, run sequentially on the same tile and can pass along data via window-based shared memory, where the output of the first becomes the input of the second. Similarly, when interdependent kernels run on neighboring tiles, they can read directly from each others memories. However differently from tile-sharing kernels, for inter-tile transfers double buffered windows are used to reduce wait time between the kernels with a maximum of 16-KiB since it has to fit within one memory module. Kernels that are non-neighboring are forced to transfer the window over a stream to the other tile. For all three cases are also possible to move data using streams, but for the cases of tile-sharing and neighboring kernels, the window-based method has 8 times higher

[1]The term"window" has been replaced with "buffer" in the programming API from Vitis version 2023.1 onwards

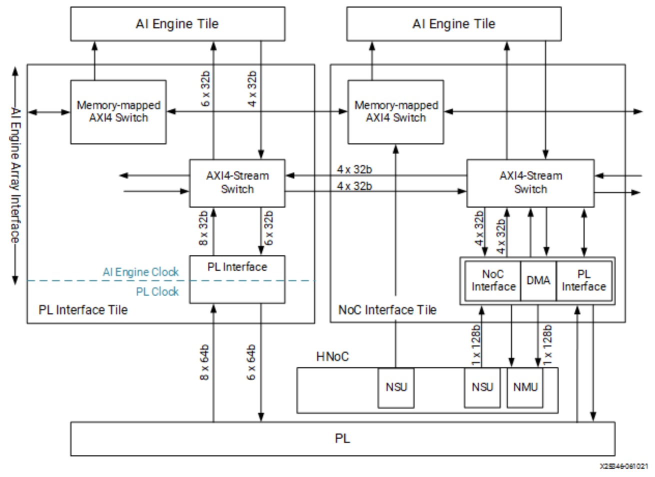


Fig. 7. Diagram of the PL and NoC interface tiles [54]

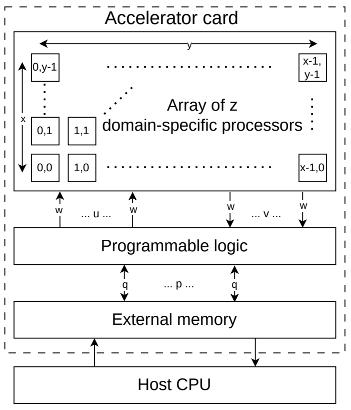


Fig. 8. Overview of general architecture

bandwidth than streams, due to the larger bit width for accessing memory. For non-neighboring kernels, stream and window based movement will have similar performance. So for window-based kernels, location is very important to obtain the maximum bandwidth.

The AIE array can interact with other parts of the system via the interface tiles at the bottom of the array, see Figure 7. There are two types of interface tiles. First there is the NoC interface tile. This tile connects the AIE to the Network-on-Chip (NoC), which mostly is used to load the program memories of the AIEs. This tile also has a Direct Memory Access (DMA) module, which lets an AIE directly access the off-chip DDR memory. There are two 32-bit streams going into and coming from the DMA and one 128-bit stream going from the DMA to the NoC Master Unit (NMU). The data from the DDR is then returned over the NoC and via a NoC Slave Unit (NSU) it is returned to the AIE array. Secondly, there is the PL interface tile, via which the AIE array can directly communicate to the programmable logic. There are eight 64-bit streams going from the PL to the AIE and 6 64-bit streams coming back. The PL and AIE are in two different clock domains and so careful consideration should be taken so that the data rates of both sides match. So the following should hold to rate match between the two components: $f_{\text{AIE}} \times d_{\text{AIE}} = f_{\text{PL}} \times d_{\text{PL}}$, where f is the clock frequency and d is the data per clock cycle. In essence, the NoC interface tile is just an extension of the PL interface tile, because the NoC interface tile also has the same PL interface as the dedicated PL interface tile. The connection between the AIE kernels and these interface tiles is defined in the AIE graph using the programming API, where a PL interface tile is referred to as PLIO and a NoC interface tile as GMIO.

The host program can either run on the Versal™ processing system (ARM) or on a host x86 processor. The ARM core has access to the same external memory as the PL and AIE array, whereas the x86 processor has to communicate over PCIe. For both methods the Xilinx Runtime (XRT) is used to communicate with the PL and AIE array. The XRT API can load a compiled configuration for the PL and AIE, which is stored in an .xclbin, as well as control the execution of the configuration. However note that only a subset of of the XRT API can be used by a host program running on an x86 processor (over PCIe). API functionality such as controlling AIE graph execution, setting run-time parameters and direct reading from memory by the AIE over the NoC interface tiles (GMIO) is not supported. In this case, the AIE graph is free running and the host is can only indirectly control the execution of the AIE graph by providing enough data via the PL.

The heterogeneous computer architecture of the Versal™ Adaptive SoC combines programmable logic, domain-specific processors (AI Engines) and general-purpose processing on the same SoC. Figure 8 introduces an abstract representation of this novel computer architecture to facilitate design space analysis and assessing the effect of different design points on performance. The array of domain specific processors consists of z processors organized in an array of x rows and y columns. There are u PL-to-array channels and v array-to-PL channels referred to as PLIO inputs and outputs respectively. Each of these PLIO channels is w bits wide, but these channels could also be aggregated to create less channels with a wider bit width, where two w channels can be combined into one $2w$ channel. The PL has p bi-directional memory channels available to/ from off-chip memory with a maximum width of q bits. These PL-memory channels cannot be aggregated to a larger width, like the PLIO. The SoC has f memory controllers for communication with the off-chip DDR memory modules. A PCIe interface enables communication over PCIe.

3 RELATED WORK

The PLF is used in Maximum Likelihood and Bayesian phylogenetic inference tools such as RAxML[50] and MrBayes[45]. Most of these tools have a parallelized version that implements multi-threading and the vector intrinsics available in most modern CPUs, such as Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX) instructions. An extensive review of phylogenetics and population genetics has been done by Corts and Alachiotis [12]. This work focuses on acceleration of phylogenetic computations.

Piñeiro et al. [39] present VeryFastTree, which is an improvement on the PLF implementation of the ML phylogenetic tool FastTree2. It keeps the same tree searching heuristics, but adds AVX and SSE versions to its PLF implementation. The speedup of VFT is compared with a single-thread execution of FT2 on three different sizes of datasets (small: 1000 taxa, medium: 25057 taxa, large: 331550 taxa). This results in a speedup ranging from 1.1× to 7.8×, depending on the number of threads, dataset size and vector instructions.

Flouri et al. [17] and Ayres et al. [6] developed optimized software libraries for statistical phylogenetics. The Phylogenetic Likelihood Library (PLL) [17] implements the likelihood calculation and tree data structure for likelihood-based inference. It can also make use of both SSE and AVX instructions, as well as multi-core processing using pthreads. Whereas the BEAGLE library [6] is developed for both likelihood- and Bayesian-based inference, but only implements the calculation of the likelihood-score. It can only make use of SSE and multi-threaded processing in BEAGLE is done using OpenMP. Ayres et al. [5] later compared the PLL version 2 and BEAGLE version 3.1.2, and found that on a single thread the AVX implementation of the PLL has up to 3.1× speedup compared against the SSE implementation of BEAGLE.

Other works have explored heterogeneous computing schemes to accelerate the PLF. Since tree scoring in modern likelihood-based phylogenetic inference tools take up about 85% to 95% of the execution time[2], many have proposed co-processors that calculate the PLF and let the CPU handle the tree searching. These co-processors are often implemented on either a Graphics Processing Unit (GPU) or FPGA.

Malakonakis et al. [32] investigate two hybrid systems that combine CPU and FPGA for phylogenetic inference. They implemented the RAxML algorithm on the systems and let the FPGA calculate the PLF, while the rest of the algorithm ran on the CPU. The first system they investigated is the Xilinx ZCU102 evaluation board featuring a quad-core ARM Cortex-A53 CPU with reconfigurable logic. It also contains a Mali-400 GPU, but its use was not explored. The second system is a cloud instance of AWS EC2 F1, which features an octa-core x86 processor (Xeon E5-2686v4) and an FPGA accelerator card connected via Peripheral Component Interconnect Express (PCIe). The first implementation is up to 64× faster than a software implementation on its ARM processor. Where the second system is up to 5.2× faster compared to a software implementation on its high-end x86 processor. When the first implementation is compared against this high-end CPU it is still up to 7.7× faster. This first system shows a better performance than the second, because it does not need to explicitly transfer any data, as the host CPU and the acceleration platform share the same memory.

Alachiotis et al. [3] propose a software cache controller exploiting the SPR searching method in order to optimize the data movement for PLF calculations on PCIe based accelerators. By caching probability vectors calculated by the PLF in-between SPR moves, less data needs to be transferred from host to accelerator and the next PLF calculation can start faster, reducing execution time. They also added double buffering in their implementation to create overlap in communication and calculation to better pipeline the system. Combined with double-buffering, this software cache implementation makes the hardware-based implementation about 3.5× faster, leading up to 7.8× speedup of the RAxML algorithm compared to a single thread on a AWS EC2 F1 cloud instance.

Following up on this research, Wijnja and Alachiotis [57] developed a generic and application-agnostic software caching framework that uses locality and data reuse to mitigate the data transfer bottleneck that often occurs with external accelerator cards. Their *SoftCache* framework can be transparently applied to an application without having to make changes to the algorithm, by positioning itself between the host program and the OpenCL library. The cache lines can store any size of data and the total cache size, cache organization and replacement policy can be tuned for a specific application, making the framework broadly applicable. The framework is evaluated using a performance model that incorporates results from GPU [25] and FPGA [3, 32] implementations, which found that the number of bytes transferred was reduced by as much as 89%. This resulted in a system-wide speedup of up to 1.7× for the GPU implementation and 3.5× for the FPGA implementations. Additionally, the general SoftCache framework shows the same performance increase as the application specific caching and double buffering method used by Alachiotis et al. [3], indicating that this generic framework achieves similar performance as specialized implementations.

Pratas et al. [40] investigate the acceleration potential of multiple platforms, including the GPU, for the PLF kernel of MrBayes. They optimized performance at the likelihood vector entry level by dividing the workload across threads, minimizing synchronization issues, and organizing data through global, block, and thread-level partitioning. Despite achieving significant reductions in PLF calculation times, the efficiency was hindered by CPU-GPU communication bottlenecks, leading to only 1.5× speedup on an Nvidia GTX285 GPU over a single-thread CPU version. Zhou et al. [68] built upon this foundation and extended it by overlapping CPU-GPU communication with computation, utilizing pipelining to reduce platform idleness and leveraging shared memory. This resulted in a good cooperation between the two platforms and had a speedup between 7.5× 12.6× compared to the work of Pratas et al. using a state-of-the-art quad-core system with an Nvidia GTX480 GPU. This implementation also had a 0.9× to 5.4× speedup when compared to the fastest CPU multi-core implementation at that time on the same machine.

Izquierdo-Carrasco et al. [25] also investigated the capabilities of GPUs to enhance the performance of the PLF implementation in RAxML and added their implementation to the PLL. They try to reduce the amount of data transferred between CPU and GPU by keeping the likelihood vectors in the GPU memory, as well as optimizing this memory layout for faster aligned data accesses by the GPU. Their results show that the PLF, which ran on an Nvidia Tesla C2075 GPU, is 2× faster than a highly optimized CPU implementation using AVX on an Intel i5-3550. Ayres et al. [5] also created a GPU implementation for the PLF in BEAGLE. They have targeted a large set of hardware by implementing both CUDA and OpenCL versions and even supports both single- and double-precision floating-point operations. Their focus lay on utilizing the massively parallel architecture of the GPU completely with wasting as little idle clock cycles as possible. The implementation ran on a Comet Supercomputer node with two Intel Xeon E5-2690v4 CPUs and a Nvidia GP100 GPU, which showed a 32× speedup compared to the single CPU threaded implementation with SSE.

Most phylogenetic tools use double-precision floating-points (DPs) to represent the likelihood vectors and all other model parameters. Since a 64-bit DP consists of twice as many bits than a 32-bit SP, these values have a higher accuracy, but at the cost of a higher memory requirement and can often be slower to compute with. This results in a performance-accuracy trade-off that needs to be considered.

Berger and Stamatakis [8] have conducted a thorough investigation of the performance and accuracy trade-offs between the use of SP and DP for the RAxML implementation of the PLF. They computed numerically sensitive operations in DP, such as the computation of the transition probability matrices ($P(t)$), and casted them to SP afterwards, but the main bulk of the PLF calculation was done with SP. This resulted in a 50% memory reduction as SP only need half of the memory compared to DP, but the SP implementation was on average 2 times slower than the DP implementation on multiple single-gene DNA datasets ranging from 150 to 1908 taxa. This is mostly due to the order of magnitude more scaling events needed by the SP implementation. However, on a large phylogeny of protein data, the SP implementation saw a 1.4× speedup compared to DP on the same SUN x4600 system using 16 cores. Lastly, they also assessed the numerical stability of the complete RAxML implementation under SP and found that numerical stability could not be achieved for datasets with more than 2000 taxa.

Bao et al. [7] proposed another improvement based on the work of Pratas et al. and Zhou et al.. They introduced an adaptive algorithm that determines the task granularity dynamically in order to reduce memory accesses, synchronization and communication overhead introduced by the fine-grain parallelism of the previous methods. This resulted in up to 63× speedup compared to a single-threaded version of MrBayes ran on a hexa-core Intel Xeon E5645 with an Nvidea GTX480 GPU. They also investigated the performance difference of their implementation with SP and DP. The DP version incurred a heavy performance penalty, especially with more alignment sites per taxa, up to the point where the SP implementation is about 40% faster than the DP version. They do however not mention any metrics about the accuracy of the two implementations and solely focus on performance gain.

The Versal™ adaptive SoCs have a novel heterogeneous reconfigurable architecture with domain specific processors. It was released in 2019[18], so it has only been around for five years at the time of writing. Therefore, there does not exist much literature on this architecture yet, especially in the domain of phylogenetics.

Zhuang et al. [69, 70], Yang et al. [65] and Xiao and Liang [58] developed frameworks for the ACAP platform that efficiently accelerates matrix multiply kernels needed for many deep learning applications. By heavily optimizing the computation-to-communication (CTC) ratio, they are able to overlap much of the communication with the multiplications and as a result maximize the off-chip memory bandwidth. Their efforts also focus on making the platform more accessible to others by abstracting away the complexities of the low-level architecture and providing a high-level interface for easily setting up a matrix multiplication accelerator.

Wijhe [56] explored the ACAP for filtering applications in radio astronomy and in particular the usability of specific mathematical libraries that are supported by the platform. They found that for their application, the libraries are often too limited in their support and careful implementation of the intrinsic operations is the only way to get the most out of the platforms capabilities.

4 SYSTEM ARCHITECTURE

The general idea of our accelerator is to use the AIE array on the Versal™ Adaptive SoC for the matrix multiplications needed for the PLF calculations (Section 4.1) and use the PL for data movement between device memory and AIE array, memory layout transformations and numerical scaling (Section 4.2). These two parts combined are referred to as the accelerator platform, which together with a host program forms the system. The host program moves the data between the host memory and the device memory and controls the execution of the accelerator platform (Section 4.3). A high-level overview of the system is illustrated in Figure 9.

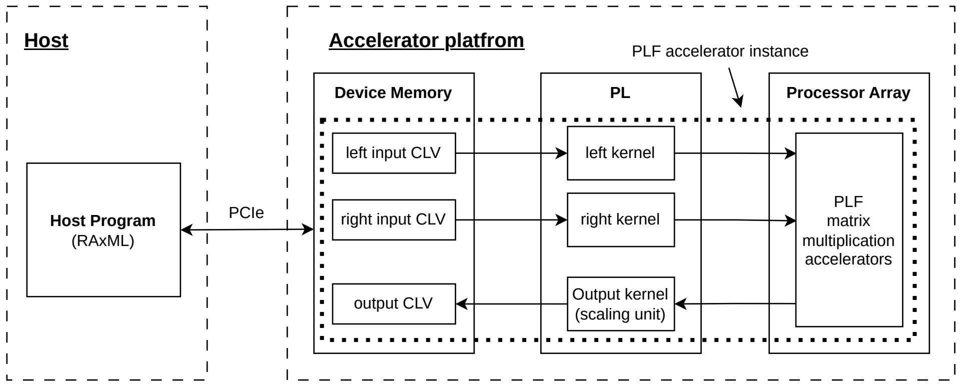


Fig. 9. High-level system overview with one accelerator instance

4.1 Domain-specific Processor Array

The PLF calculation is mapped to the AIE array by splitting the calculations into separate kernels each running on their individual AIE tile in the array. The calculation for one discrete rate is mapped to the AIE array via three distinct kernels (Figure 10). `mmul` multiplies a CLV with its corresponding SPM, which is used twice, once per child. The result of the two `mmul` kernels is combined by the `combine` kernel via an element-wise multiplication. Another matrix multiplication kernel, `ev`, multiplies the combined output with the eigenmatrix. So, the PLF calculation for one discrete rate is divided over four kernels (2 `mmul` + 1 `combine` + 1 `ev`), meaning that the PLF calculation under the Γ-model with four discrete rates, is subdivided into 4 (*rates*) × 4 (*kernels per rate*) = 16 kernels. The kernels for one PLF accelerator are combined into a graph, which is an organizational structure of the AIE API. The graph specifies the connections between the kernels and provides ports for communication with components outside the graph. This creates a reproducible instance of one PLF accelerator. A top-level graph comprises of multiple instances and connects them to PL interface ports that are also defined by the the top graph.

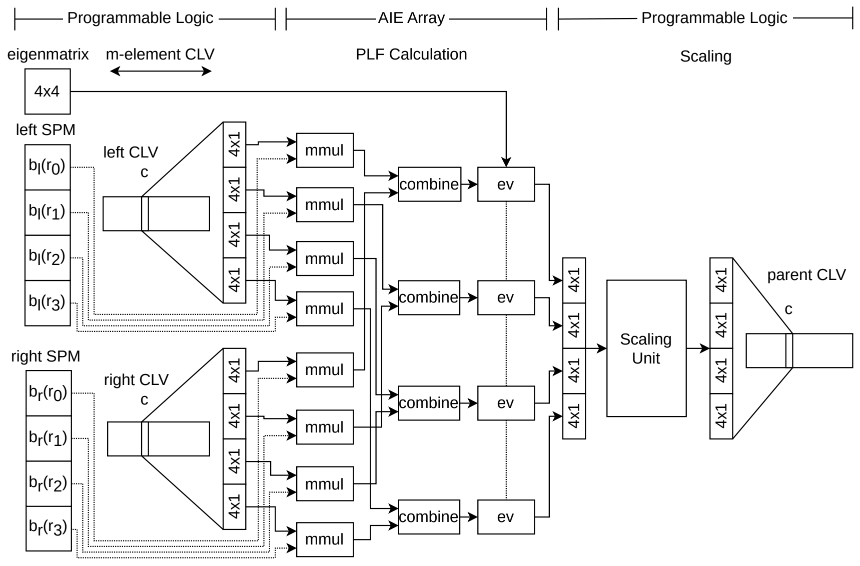
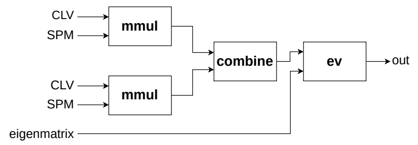
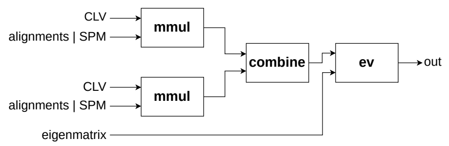


Fig. 10. Overview of the PLF computation under the Γ-model on the AIE array

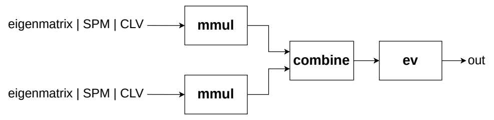
The three AIE kernels (`mmul`, `combine` and `ev`) are developed in both stream and window variants. The stream variant expects as the first value that it receives on its input to mean how many alignment patterns need to be processed and uses this value as the iteration limit in the loop around the calculation. The `mmul` kernel receives this value first and propagates it to the `combine` and `ev` kernels. The combine kernel receives streams from both the left `mmul` and right `mmul` kernel, so needs to pull the value from both streams. The window variant has to restart the kernel every time it needs a new window, so therefore the iterations of the loop containing the calculation is defined by the size of the window. The window size is a statically defined parameter of the top-level graph. Since the window-based kernels need to restart constantly, the `mmul` and `ev` kernels do not retain their SPM and eigenmatrix respectively. These matrices need to be resend for every new window. The stream-based implementation, on the other hand, runs once per accelerator call and will only needs to receive the matrices once and keep them in their registers throughout the PLF execution.



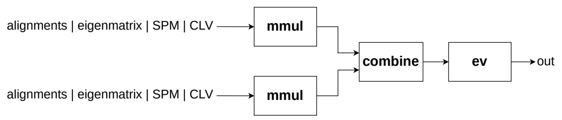
(a) Window-based with separate PLIO input layout



(b) Stream-based with separate PLIO input layout



(c) Window-based with combined PLIO input layout


alignments | eigenmatrix | SPM | CLV
mmul
alignments | eigenmatrix | SPM | CLV
mmul
combine
ev
out

(d) Stream-based with combined PLIO input layout

Fig. 11. Four different AIE kernel configurations for one discrete rate with two inter-kernel communication variants (window-based and stream-based) and two PLIO input layouts (separate and combined)

The PLF calculation for one discrete rate needs to be provided with 5 data sources: the left and right CLVs, the left and right SPMs, and the eigenmatrix. This data is provided from the PL through PL-array channels. A straightforward mapping would provide each input source over its dedicated PLIO, which requires 5 PLIO inputs per discrete rate and 17 for the four Γ rates in RAxML (the eigenmatrix is the same for all rates and thus is broadcasted to each `ev` kernel). This PLIO layout is displayed in figure 11a with window-based communication and in figure 11b with stream-based communication. However, there is a limited amount of PLIO inputs u, and the SPM and eigenmatrix PLIOs are considerably underutilized when long alignments are processed; each matrix contains (both branch and eigenvalue matrices) contains 16 floating-point values, whereas each CLV contains 4 floating-point values per alignment site,with the number of alignment sites for long alignments typically being in the order of thousands to millions base pairs [4]. To improve PLIO utilization, a new set of AIE kernels is designed where data sources are concatenated and provided over a combined PLIO layout. This layout only needs 2 PLIO inputs for one discrete rate and 8 for one PLF instance using the four Γ rates as in RAxML. A diagram of the combined PLIO layout with window-based communication is shown in figure 11c and figure 11d also shows this combined PLIO layout, but using stream-based communication. This combined PLIO layout expects the top half of the ev matrix (8 values) on the left child input and the bottom half on the right child input. The `mmul` kernels pass these along to the `combine` kernel, where these two halves are merged and forwarded to the `ev` kernel. Then the branch matrices (SPM) are expected on the inputs of the mmul kernels, which load them into their registers and enter the calculation loop to process the CLVs.

4.2 Programmable logic

The data is provided to the AIE kernels via the programmable Logic (PL). There are three distinct PL kernels per instance, one that provides the left input, one that provides the right input, and another that handles the output from the AIE array. The left and right input PL kernels are very similar. Each input PL kernel provides the eigen matrix and the four SPMs that belong to its corresponding child, which are stored in the on-chip memory of the PL. The `mmul` AIE kernels expect the SPMs to arrive in transposed form, so when the they are written to the on-chip memory at startup of the kernel, the values are rearranged so they are already stored in the correct transposed order on the PL. Specific PL kernels are developed for each of the four combinations of AIE communication methods and PLIO layouts, which transfer the data in the specific way that the AIE kernels expect it.

For stream-based implementations, the number of alignment patterns is initially sent along with the SPMs and eigenmatrix, after which the PL kernel enters a loop that iterates over all CLV data. Each loop iteration reads the data of one site for all discrete rates from the device memory and groups it per discrete rate. The data belonging to a particular discrete rate is then sent to the corresponding AIE kernels. For window-based implementations, the PL uses two nested loops to transfer the branch and eigenvalue matrices required for each window. The outer loop iterates over the number of windows, while the inner loop iterates over the number of alignment patterns per window. At the beginning of each outer loop iteration, the branch and eigenvalue matrices are sent.

The result from the PLF calculation (parent CLV) is provided to the output PL kernel over 4 PLIO channels, one for each discrete Γ rate. The output PL kernel performs numerical scaling, when necessary, and writes the resulting CLV to device memory. Scaling is performed as follows. For every alignment pattern, c, the output kernel compares all CLV entries (probabilities) with a threshold. If all probabilities are found to be below the threshold then they are multiplied with the scaling constant. Since the Versal™ Adaptive SoC only supports single-precision floating-point arithmetic, the scaling constant E is set at 2^{32} [8], which determines the threshold at $\epsilon = \frac{1}{E}$. The output kernel returns a vector with flags $\vec{f}$, where flag $\vec{f}(c)$ indicates whether pattern c was scaled up or not. In the host program, the vector of scaling flags from the output kernel can either be element-wise summed with the $\vec{U}$ vectors $(\vec{f}(c) + \vec{U}_{\text{left}}(c) + \vec{U}_{\text{right}}(c) = \vec{U}_{\text{parent}}(c))$ for regular scaling or element-wise multiplied with the $\vec{w}$ vector that stores the total number of occurrences for each alignment pattern, where the products are accumulated in a single register $s = \Sigma_{c=0}^{m}(\vec{f}(c) \times \vec{w}(c))$ to provide fast scaling. Finally, the output PL kernel reorders each CLV entry to match the RAxML memory layout and writes the resulting CLV to device memory.

4.3 Host Program

The host program runs on the host CPU and communicates with the acceleration platform via PCIe. It instantiates a connection with the accelerator card and loads the compiled accelerator platform (AIE and PL code) onto it, after which it controls all data movement to/from the device, and initializes PL kernel parameters such as the number of alignment patterns and the window size (when windows are used). For each instance of the PLF in the accelerator platform, the host program instantiates four memory buffers in the device memory; left CLV input buffer, right CLV input buffer, parent CLV output buffer and a buffer for the scaling flags. The accelerator platform contains the maximum amount of instances that the resources allow and the host program divides the alignment patterns in equally sized chunks and assigns each chunk to a different accelerator instance. The host program is not forced to divide a PLF job over all instances, but can also target specific instances, since each instance is fully independent.

The host program controls the accelerator platform over PCIe, meaning that it can only control the execution of the PL kernels and has no control over the operation of the AIE array. The AIE kernels are self-firing and start running whenever it has enough data on its inputs and enough space for its result on the output. The host transfers data for one child to an instance in the following order: 1) eigenvalue matrix, 2) branch matrices for each Γ rate and 3) CLV entries. In the case of the separate input layout, the eigenmatrix is only send to the left child.

The host program deploys multiple threads to move the input and output buffers for all accelerator instances over different PCIe lanes, with each thread utilizing a single PCIe lane. This enables the simultaneous transfer of the left and right buffers for ingress, as well as the parent and scaling buffers for egress. Figure 12 provides a timing diagram of parallel data movement over 8x PCIe lanes, and parallel execution on four accelerator instances. A pair of threads is used per accelerator instance. Each pair blocks on a local barrier to synchronize the host and device memories after transferring the left and right input buffers. Thereafter, one of the threads in each pair initiates execution of the corresponding accelerator instance. Each pair blocks again on a local barrier until processing is over, indicated by the output PL module, before initiating the transfer of the parent and scaling buffers. Because parallel execution on multiple accelerator instances can proceed asynchronously, as threads synchronize in pairs, all threads wait on a global barrier after completing their respective buffer transfers. This global barrier ensures coherency between the device and host memories before allowing the host program to continue.

5 IMPLEMENTATION

5.1 Development platform

The accelerator platform is implemented on an AMD Versal™ Adaptive SoC, in particular the VC1902 chip on the VCK5000 development card. For development, the card in a faculty server is used. The server has 2 Intel Xeon Silver 4216 CPUs totaling with 16 physical cores each and with 256 GB of main memory using PCIe 4.0 over 8 lanes. It runs Rocky Linux 8.9, v++ 2022.2 (Vitis command line compiler), XRT 2023.2 (Xilinx Runtime) and gcc 8.5. The v++ command was used to compile the AIE graph and PL kernels,

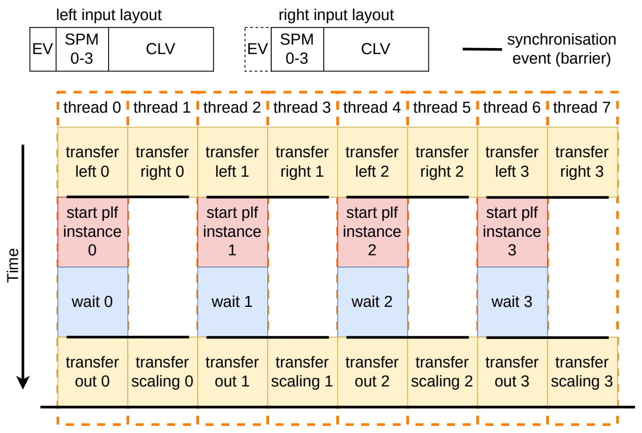


Fig. 12. Overview of parallel data transfers to/from the hardware card. Each thread uses a PCIe lane, and each thread pair (defined by local synchronization) handles IO and control of one accelerator instance. This example shows the execution for four parallel instances that each process a fourth of the total amount of alignment patterns. At the top of the diagram, the left and right buffer layouts per instance are shown. The right buffer only contains the eigenmatrix when the combined layout is used (note the dotted frame). The SPM indices refer to the discrete Γ rates.

as well as linking them to create an .xclbin file to be loaded by the host program. The host program is developed in C++ and interacts with the accelerator card via the XRT libaray API.

The VC1902 has $z = 400$ AIEs in an array of $x = 8$ rows and $y = 50$ columns [13], where the letters refer to the parameters of the model presented in Figure 8. These processors run at a frequency of 1.25 GHz as defined by its speed grade *-2M*[14]. The array is connected to 39 PL-interface tiles, which results in $u = 39 \cdot 8 = 312$ streams from the PL to the AIE array and $v = 39 \cdot 6 = 234$ streams from the AIE array back to the PL. These streams have bit width of $w = 32$, but they can also be aggregated to a maximum of $u = 156$ and $v = 117$ for $w = 64$ or a maximum of $u = 78$ and $v = 58$ for $w = 128$. The PL has a maximum clock frequency of 625 MHz, meaning that the connection from PL to the array has a maximum throughput of 1.56 TB/s and the maximum throughput back is 1.17 TB/s [14, 54]. Similarly the array also has access to 16 NoC-interface tiles, which could support up to a 160 GB/s connection directly between the array and device memory. However, these connections are not supported for a host program controlling the Versal™ over PCIe.

The PL has a single continuous region of about 900K Look-up Tables (LUTs) and about 1.7M Flip-Flops (FFs). The PL also has access to 1968 Digital Signal Processers (DSPs) and on-chip memory divided into 27.5 MB distributed LUTRAM, 967 Block RAM blocks totaling 34 MB and 463 UltraRAM blocks of a total of 130 MB. By experimentation it was found that the PL has a maximum of $p = 30$ logical connections to the off-chip memory. The PL can interface to the off-chip memory with a maximum bit width of $q = 512$ bits. The off-chip memory is provided by the VCK5000 that the VC1902 chip is mounted on and consists of 4 modules of 4GB DDR4-3200 memory. Each module has a maximum operating speed of 25.6 GB/s and of the four modules, one module is reserved for the Prossessing System (PS) that consists of a dual core ARM Cortex-A72. This leaves 3 modules with a total of 12 GB for use by the PL and AIE array. DDR4 memory only has a 64-bit data width, but can run much faster than the PL, so one PL memory access of 512-bits is provided by multiple memory accesses of the DDR4 module.

5.2 AIE array

A PLF accelerator instance uses 16 AIE kernels, with each kernel pinned to a dedicated AIE tile. This means that a maximum of 25 PLF accelerator instances fit onto the AIE array. However, the limited number of PLIO channels, which connect the PL and the AIE array, limits system scaling. A PLF acceleration instance using the separate input layout for the kernels uses 17 PLIO inputs and 4 PLIO outputs, whereas an instance using the combined input layout uses only 8 PLIO inputs and also 4 PLIO outputs. The kernels operate with 128-bit-wide input and output, which can move one site of CLV entries for one discrete rate of DNA in a single clock cycle. So, 78 aggregated PLIO inputs (u) are available, from which can be deduced that only $\lfloor \frac{78}{17} \rfloor = 4$ instances can be created with the separate input layout and $\lfloor \frac{78}{8} \rfloor = 9$ instances using the combined input layout. That means that the PLF kernels only use 16% and 36% of the AIE array, respectively, showing that the PL-to-AIE channels are the main resource constraint for mapping the PLF to the VC1902 SoC.

Table 1 shows the amount of cycles each kernel takes as estimated by the AMD *aiesimulator* tool. It only includes the kernels that use windows to transfer the data, because the for loop in kernels using streams have a variable limit, set during runtime by the first value the kernel receives. Therefore the tool cannot make an accurate estimation for how many cycles the stream implementations need. There are three

Table 1. Execution time (clock cycles) reported by *aiesimulator* per AIE kernel for the window implementations with the separate (col. "Sep.") and combined (col "Com.") input layouts.

Size	mmul		combine		ev	
(KiB)	Sep.	Com.	Sep.	Com.	Sep.	Com.
1	266	258	66	67	266	254
8	1834	1826	290	291	1834	1822
16	3626	3597	546	544	3626	3626

window sizes for each input layout (small: 1KiB, medium: 8KiB and large: 16KiB), which show a linear increase of execution time for increasing window size. For kernels with larger windows the percentage of startup/shutdown overhead is smaller, and therefore the execution time for one kernel with a 16KiB window is shorter than sixteen executions of a kernel with a 1KiB window. There is barely any difference in execution time between the kernels for the separate and combined layouts, indicating that the there is no performance penalty in the combined layout for sending the branch and eigenvalue matrices over the same PLIO as the CLV data.

As mentioned before, the AIE array is used 16% for the separate layout and 36% for the combined layout when only taking the AIE kernels into account. However, some AIE tiles are only used to hold windows in their memory for kernels on neighboring AIE tiles. When these tiles are also included in the percentage of resource utilization, then the separate layout (4 instances) sees 23%, 25% and 34% utilization for the 1KiB, 8KiB and 16KiB implementations, respectively, while the combined layout (9 instances) uses 41%, 62% and 76% of the array for those respective window sizes. The stream implementations only use the memory for the AIE kernel code, which resides on the same tile as the kernel and therefore has the array utilization dictated by the AIE kernels. It is to be noted that every implementation has been mapped automatically to the array, with the exception of the 9 instances using 16KiB windows and the combined layout, where the mapper needed to be assisted to find a mapping. This manual mapping makes more effective use of the available resources, indicating that the AIE array utilization for kernels with smaller window sizes could also be reduced with a manual mapping.

5.3 Programmable Logic

A PLF instance on the AIE graph is provided with data by two input kernels that are implemented on the PL and its output is handled by the output kernel. These PL kernels are described using Vitis HLS (High-Level Synthesis) and compiled with the v++ command. The kernels interface with a 512-bit port, which perfectly fits the 16 values for one site of DNA data with four discrete rates or the 4×4 branch and eigenvalue matrices. Each kernel also has four 128-bit AXI4 stream interfaces to connect to each discrete rate input/output of the AIE graph, which are connected during the linking at build time. The kernels used for the separate layout also has another set of four stream interfaces for the branch matrices and the left input kernel has an extra stream interface for the eigenmatrix. The kernels receive their parameters (pointer to device memory, number of alignments and window size) over AXILITE. The output kernels have an extra parameter for a second memory address where it can write the scaling administration to.

The AIE array runs at a fixed 1250MHz, while the clock of the PL can be scaled. Ideally the data rate of both should be matched. The PL kernels read/write four times as much data per clock cycle over the

Table 2. Resource utilization for every variant of the PL kernels for clock frequency $f_{PL} = 250$MHz

	Window						Stream					
	Separate			Combined			Separate			Combined		
	Left	Right	Out	Left	Right	Out	Left	Right	Out	Left	Right	Out
LUT	13346	12727	12515	13118	13118	12515	11711	11605	10651	12414	12414	10651
FF	10222	9517	9819	9969	9969	9819	8099	7965	8321	8286	8286	8321
DSP	4	4	22	4	4	22	0	0	16	0	0	16

streams compared to theAIE array. So, ideally the PL runs at ≥ 312.5MHz to keep the AIE array constantly provided with enough data. However, the current design can only go up to 250 MHz without having a timing violation. This slightly reduces the performance of the system.

For the different configurations of the AIE graph, there are four different sets of PL kernels. Table 2 shows the difference in resource utilization for each of the four sets at PL frequency $f_{\mathrm{PL}} = 250$MHz. It shows that the three PL kernels needed for one instance collectively utilize $< 4.5\%$ of the LUTs, $< 2\%$ of the FFs and $< 1.5\%$ of the DSPs. The reason for the slight difference in resource utilization is that in the case of a separate PLIO layout, only the left kernel reads and sends the eigenmatrix. That is also why the left and right kernels have the same resource utilization for the combined PLIO layouts. Although they are not identical, because they each send a different half of the eigenmatrix. The implementations supporting a window based AIE graph use slightly more resources, because it uses an outer and inner loop and send the branch and eigenvalue matrices for every instance of the outer loop. The stream-based implementation is slightly simpler, because it only sends the matrices once at the start of execution and then enters a single loop that iterates over the alignment data in one go. The output kernels are identical for the separate and combined versions and therefore use the same resources.

6 EVALUATION

6.1 Design-space exploration

A two-step design space exploration based on a series of experiments is performed to gauge system performance and identify bottlenecks. The performance is reported as the average throughput, measured in CLV Entries per Second (CLVES), over 100 repetitions per measurement. A CLV entry is 16 single-precision values under the 4-category Γ model, requiring 400 floating-point operations (RAxML implementation). Thus, CLVES throughput can be converted to FLoating-point OPerations per Second (FLOPS) by multiplying by 400. The throughput of the AIE array is assessed with 1, 2, 4, 8, and 9 (when possible) accelerator instances for a thousand (1K) to ten million (10M) alignment patterns. Both stream and window based variants are tested, where the window-based variant is implemented with a small (1 KiB), a medium (8 KiB), and a large (16 KiB) window to assess the impact of the window size. The evaluation is done on a server of the Heterogeneous Accelerated Compute Cluster (HACC) at ETH Zürich . It has two AMD EPYC 7V13 CPUs (64 physical cores each), and 512 GB RAM, as well as two VCK5000 cards, each connected via 8-lane PCIe. It runs Ubuntu 20.04.6 LTS, XRT 2022.2, and gcc 9.4. The xclbins were built on the development server and copied to the evaluation server, where only one of the VCK5000 cards was used for evaluation.

Table 3. Attained throughput (in 10^6 CLVES) of up to 9 accelerator instances (when possible) when input (CLVs) is generated on-chip in the PL (col. "PL") and when fetched from device memory (col. "MEM"). This table does not include any PCIe overhead. Columns "MEM" include ingress/egress buffering via the PL and output scaling and writing to device memory.

Configuration	Accel.	1K		10K		100K		1M		10M	
(interface, layout)	inst.	PL	MEM	PL	MEM	PL	MEM	PL	MEM	PL	MEM
stream, separate	1	104.8	20.9	141.9	89.6	147.3	137.0	147.9	145.6	147.9	147.5
	2	130.2	16.7	259.8	110.6	290.5	194.5	295.4	174.2	295.8	268.7
	4	137.1	20.2	363.6	112.5	563.9	280.0	588.5	338.1	591.3	315.0
stream, combined	1	102.4	20.2	142.4	76.0	147.3	108.0	147.8	109.8	147.9	114.1
	2	133.2	15.6	267.5	95.7	291.2	189.1	295.5	211.8	295.8	224.8
	4	115.5	13.9	415.5	95.0	560.9	299.3	589.1	270.7	591.4	408.6
	8	90.9	9.4	512.8	57.8	1008.5	559.5	1166.9	490.9	1181.4	566.6
	9	88.3	10.4	615.7	68.3	1184.6	504.5	1309.3	520.2	1328.3	583.1
1-KiB window, separate	1	149.0	20.6	215.3	95.7	227.5	159.0	228.4	137.1	228.5	170.0
	2	163.6	19.2	361.0	128.7	448.2	256.1	456.3	221.4	457.0	306.7
	4	137.2	17.3	579.3	120.5	837.9	255.3	908.2	332.6	913.6	243.3
1-KiB window, combined	1	150.6	21.1	216.0	84.4	227.2	124.3	228.4	130.2	228.6	130.9
	2	155.8	17.4	398.1	101.9	449.8	220.4	456.3	228.0	457.0	260.3
	4	156.3	14.1	581.2	96.8	833.0	316.0	907.7	345.2	913.6	248.0
	8	117.6	13.2	643.6	91.1	1408.5	575.9	1780.8	598.3	1823.5	582.5
	9	81.2	6.6	602.1	68.2	1538.3	684.7	1996.4	583.5	2050.3	1003.4
8-KiB window, separate	1	153.9	22.9	234.4	78.3	245.5	108.9	246.9	115.8	247.1	115.0
	2	93.9	19.3	316.4	95.6	483.6	143.6	493.1	166.2	494.0	201.0
	4	102.1	15.9	642.8	110.4	907.5	239.2	976.7	364.1	987.1	233.0
8 KiB window, combined	1	150.1	22.1	228.9	86.3	245.5	131.4	246.9	137.3	247.1	140.4
	2	148.3	19.7	426.0	108.9	483.2	232.8	492.9	266.9	494.1	271.2
	4	164.8	13.1	604.1	96.9	909.0	386.0	980.7	396.1	987.6	467.5
	8	86.0	8.8	593.5	67.8	1565.4	658.6	1925.0	672.9	1971.5	913.9
	9	85.2	6.3	648.4	56.4	1651.5	703.3	2149.0	806.9	2216.4	898.1
16-KiB window, separate	1	145.7	18.1	232.4	73.1	246.9	108.4	248.3	114.4	248.5	115.1
	2	187.4	14.7	422.6	93.2	488.1	175.4	496.0	125.3	496.9	134.9
	4	131.3	10.6	533.0	89.3	909.8	280.2	988.3	227.6	993.2	226.1
16-KiB window, combined	1	138.1	18.4	225.9	79.3	246.9	118.8	248.4	125.3	248.5	121.2
	2	94.7	14.2	423.4	96.6	483.6	200.0	495.7	231.7	496.9	246.3
	4	79.3	10.3	627.0	82.2	928.3	313.6	985.2	383.0	992.7	374.9
	8	59.4	6.1	643.9	54.1	1382.3	380.9	1922.8	509.1	1980.6	494.2
	9	63.6	5.4	651.0	51.6	1599.2	456.0	2141.1	539.3	2228.9	532.4

6.1.1 Chip performance.

In the first step, the aim is to quantify the performance of the VC1902 chip apart from the VCK5000 platform. Modified the input PL kernels generate the input data instead of reading from external device memory. They repeatedly transfer a single CLV entry stored in on-chip PL memory. The output PL kernel was modified not to perform scaling or write to external memory. The host program was also modified to only initiate the process and measure execution time using *std::chrono::steady_clock*. For this test, the XRT library only starts the PL kernels once and times 100 consecutive runs to reduce XRT-incurred synchronization overheads and better approach the performance of the VC1902 chip.

Column "PL" in Table 3 provides the attained PLF throughput on the VC1902 chip apart from the VCK5000 platform, reaching up to 2.2×10^9 CLVES. It can be observed that throughput is heavily dependent on the CLV length, with short-CLV runs dominated by the XRT communication overhead.

Multiple accelerator instances improve performance for long CLVs, due to a more favorable computation-to-communication ratio, but do not scale beyond four instances for short CLVs. The performance of one instance, which processes one CLV entry per cycle, is limited by the PL frequency (250×10^6 CLVES due to $f_{PL} = 250$ MHz). The PLF design on the AIE has a theoretical maximum of 400×10^6 CLVES: each tile can perform 8 FLOPs per cycle operating at 1250 MHz, resulting in 10 GFLOPS per tile, and 16 tiles are used by each accelerator instance. RAxML performs 400 FLOPs per CLV entry, which gives $\frac{16 \text{ (tiles) } \times \text{ } 10{\times}10^9\text{(FLOPS)}}{400 \text{ FLOPs/CLV\_entry}} = 400 \times 10^6$ CLVES. Thus, the system could benefit from a design that allows for a higher PL frequency. However, the stream implementation only achieves throughput of up to 150×10^6 CLVES, likely due to the 8x smaller bit width communication compared to the window implementation. The window implementation, on the other hand, incurs less overhead for resending the SPMs and eigenmatrix as the window size increases, converging to the maximum attainable performance dictated by the PL frequency. The comparable performance of the two input layouts for the same interface suggests that the combined layout is preferable, as it enables more parallelism (more instances) without introducing any I/O overhead.

6.1.2 Platform performance.

In the second step, the complete functionality of the input/output PL kernels (device memory access, CLV permutation, numerical scaling) is implemented. The measurements are repeated, this time including accessing device memory to quantify performance of the VCK5000 platform. Column "MEM" in Table 3 shows the results. For a single instance, the stream implementations achieve nearly the same throughput as in the previous test (column "PL"), whereas the window implementations only reach around 60%. Yet, the window implementations are generally faster, as they are only limited by memory, while the stream implementations are still limited by the AIE array. A more profound difference can also be observed between the input layouts than in the "PL" test. For streams and small windows, the separate layout is faster because it allows CLV computations to start sooner by loading the two SPMs and the eigenmatrix in parallel. For the medium and large windows, the combined layout is faster because the SPM and eigenmatrix loading time is negligible compared to loading the input CLVs. When processing short CLVs, performance is worse than in the "PL" test. It is speculated that the XRT communication overhead is a major limiting factor in this case. Longer CLVs rely less on XRT communication but stress the memory more, making memory bandwidth the performance bottleneck.

6.2 System performance and scalability

The design-space exploration informs that the combined input layout coupled with a medium window size results in the best-performing design point. Next, the overall system performance is evaluated using the best-performing design point and implementing the complete functionality of the host program (data movement over PCIe), measuring throughput for up to 10M patterns.

Figure 13 shows complete system performance of one instance using 8-KiB windows and the combined input layout ("PCIe (sequential)" in the figure). Each instance uses two PCIe Gen. 4.0 lanes, providing each instance with 4GB/s full-duplex transfer speed. With the PLF requiring 128 bytes per DNA alignment site (64 bytes per child), the theoretical maximum bandwidth is 31.25×10^6 CLVES. The sequential accelerator execution (input transfer $\rightarrow$ execution $\rightarrow$ output transfer) approaches this limit for increasing CLV lengths. A performance model estimates the system performance ("PCIe (optimized)" in the figure) incorporating

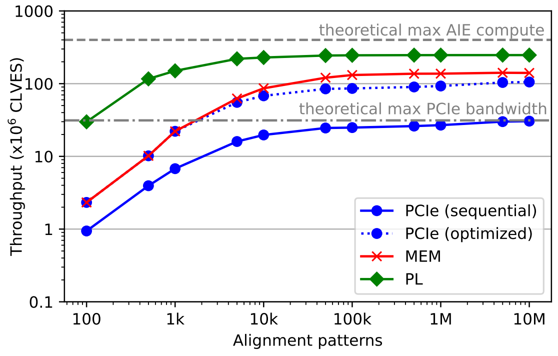


Fig. 13. System performance with one accelerator instance using 8-KiB windows and the combined input layout. The dotted line shows a projected performance of the complete system with optimized data movement as proposed by [3, 57].

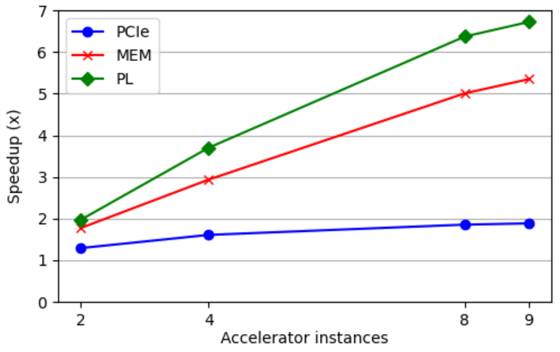


Fig. 14. Speedup of multiple instances using combined input and 8-KiB windows for 100K patterns: chip (green/diamond), platform (red/cross), and complete system (blue/dot).

optimizations for data movement over PCIe (caching [57] and double buffering [3]), resulting in 3.45× faster processing than the sequential accelerator execution. It was observed that processing short CLVs (up to 10K) is limited by XRT communication overhead, whereas for longer CLVs, throughput exceeds the theoretical limit of PCIe for sequential execution and approaches the performance of the accelerator when accessing data in device memory ("MEM"). The figure shows that the system is IO-bound. The maximum attainable performance of the AIE array is lower than its theoretical peak, limited by the operating clock frequency of the PL ("PL" line in the figure). Longer CLVs (> 10K) demonstrate the performance limitations of data accessing in device memory ("MEM" in the figure) and data movement through PCIe ("PCIe (*)" in the figure), while performance for short CLVs is reduced due to XRT communication overheads.

Figure 14 shows how the implementation scales to multiple accelerator instances using an 8-KiB window and the combined input layout for a CLV of length 100K (average-case scenario). As the number of instances increases, the AIE array ("PL") scales linearly, whereas accessing device memory ("MEM") reduces performance to about 75%. The platform has three DDR modules that are shared among the multiple instances (each instance uses three memory channels intensively and one channel very lightly). Furthermore, since the input data and the output data of each instance is accessed in different memory spaces, the memory accesses are poorly coalesced, further reducing performance. Including data movement over PCIe ("PCIe") reduces scaling performance considerably, indicating that the host-device transfer bandwidth has a major impact on system scalability.

6.3 Performance comparison

The presented system is compared with other FPGA [32] and GPU [25] accelerators for the PLF, and the best-performing (to the best of the authors' knowledge) CPU implementation, i.e., the multi-threaded AVX2 implementation of RAxML v.8.2.12, executed on both the Xeon-based development server (Section 5) and the EPYC-based HACC server.

Figure 15 presents a performance comparison with RAxML (based on the average of 100 PLF calls) using 1 and 16 CPU cores for short (10K) to long CLV lengths (10M) on both servers. The input datasets for this comparison were simulated using *seq-gen* v.1.3.2 [47]. On both systems, RAxML scales almost linearly up to 16 cores, as illustrated by figure 16. As the number of threads increases, accessing memory becomes the bottleneck, and the effect of caching is more profound, since RAxML benefits from locality of reference when accessing CLVs during consecutive SPR moves [3]. The Xeon server has 44 MB of cache memory that can store up to 70 short CLVs, 7 average-length CLVs, but only a fraction of one long CLV, explaining why its 16-core throughput performance with longer CLVs (100K, 1M, and 10M) is lower than with the short CLV length (10K). The EPYC server maintains consistent performance up to 16 cores for all CLV lengths, because of its considerably larger cache memory size (584 MB). For more than 16 cores, only the shorter CLV lengths ($<$ 1M) continue to scale, whereas the long CLV lengths use all of the cache making the throughput plateau as the cores cannot be provided with data fast enough anymore.

The sequential execution of the presented accelerator system is up to 1.5× faster and 3.0× faster than one EPYC core and one Xeon core, respectively. For short CLVs, a slowdown between 0.5× and 0.7× is observed against both CPUs (one core). The overall performance of the accelerated system with PCIe optimizations is between 2.4× and 5.3× faster than the EPYC core, and between 3.8× and 10.6× faster than the Xeon core. The optimized system shows comparable performance with 16 Xeon cores for long CLV lengths (between 0.9× and 1.3×), but 0.3× slowdown is observed for short CLVs. The 16 EPYC cores, on the other hand, are on average 3× faster than the system with optimized movement. When the accelerator accesses data in device memory only, to provide insights into the potential system performance from RAxML running on the Versal™ processing system, between 2.8× and 23.8× faster processing than one EPYC core can be observed, and up to 2.0× faster processing than 16 cores for the 10M-long CLV. With respect to the Xeon CPU, between 4.2× and 47.0× faster processing than one core can be observed and up to 4.0× faster processing than 16 cores for the 10M-long CLV. Since the accelerator platform can still achieve speedup over 16 cores using AVX2, it shows that the data movement over PCIe is a major bottleneck in the system, especially considering that for 1M-long and 10M-long CLVs the accelerator platform is still 1.2× and 1.9× faster than 128 high-end CPU cores, respectively (see figure 16). For the short 10K-long CLV, implementations with

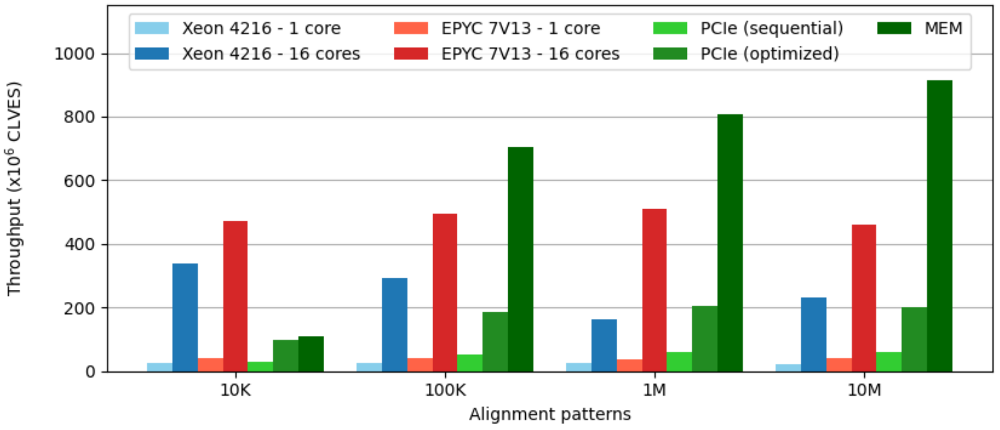


Fig. 15. Throughput comparison of the Versal-based system (PCIe sequential and optimized) and platform (MEM) performance with multi-threaded, AVX2 RAxML execution.

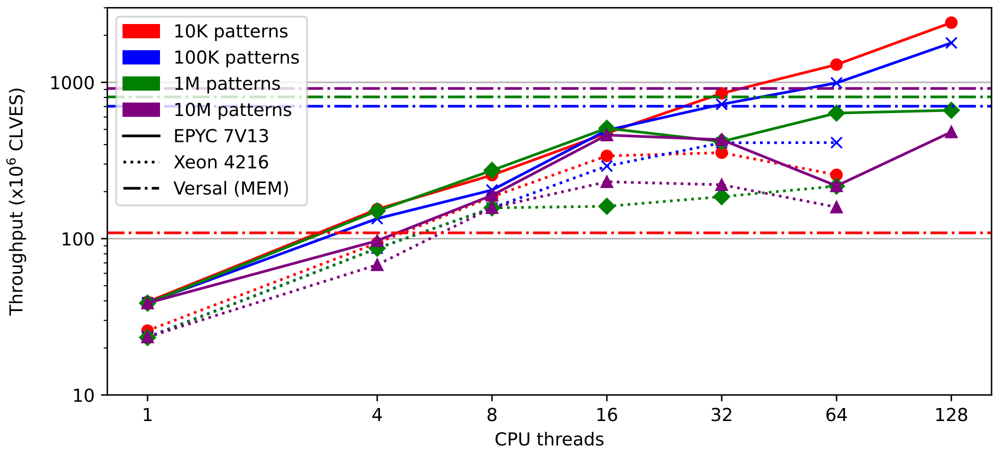


Fig. 16. Throughput of the RAxML CPU implementation of the PLF using AVX2 for increasing CPU threads of the Xeon (32 cores/64 threads) and EPYC (128 cores/128 threads) systems. The systems are measured for CLV lengths of 10K (red/dot), 100K (blue/cross), 1M (green/diamond) and 10M (purple/triangle). Horizontal lines show the throughput of the accelerator platform that only accesses device memory ("MEM").

more than 4 CPU cores are already faster than the accelerator platform, with the platform only reaching 40% of the throughput that 16 EPYC cores can achieve and 60% compared against 16 Xeon cores, indicating the impractical use of hardware acceleration for such a small workload.

Although not a fair comparison, it is to be noted that the computational potential presented by the "PL" columns in table 3 indicates that, even when limited by the PL frequency, the PLF AIE kernels could achieve 1.4× faster execution for short 10K-long CLVs and up to 4.8× faster execution for the 10M-long CLVs than 16 high-end EPYC cores if the kernels had access to sufficient memory bandwidth. Considering that the current implementation only utilizes 36% of the available compute resources of the AIE array, solidifies the notion that the application is IO bound and the platform has still much computational potential left.

The presented implementation is also compared to other FPGA-based PLF accelerators [32] mapped to a ZCU102 evaluation board, with the host program (RAxML) running on the ARM processor of the Zynq SoC, and an AWS EC2 F1 instance, with the host program running on an Intel Xeon E5-2868v4 processor, transferring data over PCIe. The Versal™ accelerated system (with optimized PCIe transfers) is between 1.9× and 5.1× faster than the special-purpose architecture on the ZCU102, and up to 21.9× faster in terms of the computational capacity of the AIE array (local access only). This speedup could be achieved if the Versal™ ARM processor was utilized in the same way as the Zynq ARM processor is used in the ZCU102 system [32]. In comparison with the cloud accelerated system (F1), which maps the PLF accelerator to a Virtex VU9P FPGA card, the sequential accelerator execution on the Versal™ is between 2.0× and 3.2× faster, where this speedup is projected to improve to between 6.9× and 11× with the optimized data movement.

A direct comparison with the GPU solution of Izquierdo-Carrasco et al. can be misleading, as the PLF implementation of the Phylogenetic Likelihood Library [17] was used as the reference for acceleration and evaluation. The authors reported slowdowns for less than 10k alignment patterns, and speedups up to 1.8×

for 100K patterns, when compared to one Intel i5-3550 CPU core using AVX intrinsics. The PLL, however, is derived from the RAxML implementation, for which the accelerated system presented in this thesis achieves up to 8.9× faster execution than 1 Xeon CPU core using AVX2 intrinsics.

7 CONCLUSION AND FUTURE WORK

This work presents a comprehensive design-space exploration to identify bottlenecks and limitations of the novel Versal™ architecture in accelerating the most commonly used operation in large-scale phylogenetic analysis: the Phylogenetic Likelihood Function. This exploration indicates that the accelerator with window-based inter-tile communication performs better than the stream-based implementation. Furthermore, from the results can be concluded that the minimal extra overhead from concatenating low bandwidth data sources to high bandwidth data sources has a negligible effect on accelerator instance performance and allows for increased device utilization as a smaller number of PLIO channels between the PL and AIE are used per instance, enabling the deployment of more parallel instances. Therefore concatenation of data sources is preferred. Comparison with the RAxML implementation of the PLF for processing DNA data shows that the system can achieve up to 10.6× speedup over a single modern CPU core using AVX2 instructions, whereas only comparable results or slowdown is observed against 16 cores. When comparing the CPU implementation with just the accelerator platform (only considering local memory access), it becomes clear that the device memory and PCIe data movement form a severe bottleneck in the system, as the accelerator platform can still achieve up to 1.9× faster execution for long CLVs compared with 128 high-end EPYC cores using AVX2, while utilizing just 36% of the Versal™ AIE compute resources. The bottlenecks formed by device memory access and PCIe communication indicate that only high arithmetic intensity kernels can fully utilize the substantial computational power of the AIE array on the Versal™ SoC, which could, theoretically, compute the PLF up to 5.3× faster than 128 EPYC cores and 130× faster than a single Xeon core, both using AVX2 (assuming 100% compute utilization and sufficient PLIO channels).

As future work it would make sense to implement the protein-based version of the PLF, since RAxML needs 9680 FLOPs per CLV entry for the 20-state protein data, meaning that the protein implementation would need $\frac{9680}{400} = 24.2\times$ more compute resources than the DNA implementation for only $\frac{20}{4} = 5\times$ more data movement. The protein-based implementation has therefore nearly 5× higher arithmetic intensity than the DNA-based implementation and can likely lead to higher device utilization and more significant performance advances. Furthermore, some areas have not been touched by the design-space exploration, such as the cascade interfaces in the AIE array. The cascades could move up to 8 values per clock cycle, therefore it would be interesting to explorer a hybrid solution with cascades and windows to reduce the number of AIE tiles used just for holding a buffer, to enable placement of many parallel high arithmetic intensity kernels. Also, the presented exploration has not established the performance of the AIE array in isolation as the results in the“PL” column in table 3 are limited by the PL frequency. This means that PL kernels that have a shorter critical path, thus allowing for a higher PL clock frequency, can improve performance, but it is unclear by how much. The following design iteration of the PL kernels should, next to improved critical path, investigate a monolithic design instead of the modular design presented in this work. The monolithic design will distribute the CLV site-by-site instead in chunks over the instances, which will not effect correctness since the PLF can be computed independently between sites, but it may improve memory coalescence and reduce memory congestion as only one kernel accesses memory, preferably in bursts.

This design would also simplify the host program as there is less data preparation on the host side. To circumvent the bottlenecks caused by the device memory and PCIe transfers, it should be considered to exploit the two 100Gbit Ethernet ports of the VCK5000 to directly stream data to the PL, which could improve the host-device throughput from 125MA/s (8GB/s) to 195 MA/s (12.5GB/s), considering half of the total bandwidth available to send each child in parallel. Moreover, a comprehensive comparison should be performed to understand the performance trade-off between running the search algorithm of RAxML (host program) on the integrated ARM CPUs of the Versal™ SoC (to benefit from local data access) and running it on a high-end x86 CPU (which requires data movement over PCIe). Finally, the comparison between the Versal™ and CPU implementations could be aided when energy efficiency is taken into account, especially where the two system show similar performance, as phylogenetic analyses are often multi-hour computations.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude towards my supervisor, Dr Nikolaos Alachiotis, for his invaluable feedback and support he gave during my research. His guidance helped me throughout the entire research and writing of this thesis. I am also extremely thankful for the contributions of Dr Mario Ruiz Noguera as he helped me with any question I had about the Versal architecture, which massively accelerated the development of the system. I am grateful to ETH Zurich for providing access to their Heterogeneous Accelerated Compute Cluster (HACC), which allowed me to bring my research to the next level. I also extend my heartfelt thanks to my supportive partner, family and friends who endured my talking about weird computer things, but ultimately supported me throughout this journey.

REFERENCES

[1] Ogün Adebali, Aylin Bircan, Defne Circi, Burak İşlek, Zeynep Kilinc, Berkay Selcuk, and Berk Turhan. 2020. Phylogenetic analysis of SARS-CoV-2 genomes in Turkey. *Turkish Journal of Biology* 44, 7 (2020), 146–156.

[2] Nikolaos Alachiotis. 2012. *Algorithms and Computer Architectures for Evolutionary Bioinformatics.* Ph. D. Dissertation. Technische Universität München.

[3] Nikolaos Alachiotis, Andreas Brokalakis, Vasilis Amourgianos, Sotiris Ioannidis, Pavlos Malakonakis, and Tasos Bokalidis. 2021. Accelerating Phylogenetics Using FPGAs in the Cloud. *IEEE micro* 41, 4 (2021), 24–30. https://doi.org/10.1109/MM.2021.3075848

[4] Samuel V Angiuoli and Steven L Salzberg. 2011. Mugsy: fast multiple alignment of closely related whole genomes. *Bioinformatics* 27, 3 (2011), 334–342.

[5] Daniel L Ayres, Michael P Cummings, Guy Baele, Aaron E Darling, Paul O Lewis, David L Swofford, John P Huelsenbeck, Philippe Lemey, Andrew Rambaut, and Marc A Suchard. 2019. BEAGLE 3: improved performance, scaling, and usability for a high-performance computing library for statistical phylogenetics. *Systematic biology* 68, 6 (2019), 1052–1061.

[6] Daniel L Ayres, Aaron Darling, Derrick J Zwickl, Peter Beerli, Mark T Holder, Paul O Lewis, John P Huelsenbeck, Fredrik Ronquist, David L Swofford, Michael P Cummings, et al. 2012. BEAGLE: an application programming interface and high-performance computing library for statistical phylogenetics. *Systematic biology* 61, 1 (2012), 170–173.

[7] Jie Bao, Hongju Xia, Jianfu Zhou, Xiaoguang Liu, and Gang Wang. 2013. Efficient implementation of MrBayes on multi-GPU. *Molecular biology and evolution* 30, 6 (2013), 1471–1479.

[8] Simon A Berger and Alexandros Stamatakis. 2010. Accuracy and performance of single versus double precision arithmetics for maximum likelihood phylogeny reconstruction. In *Parallel Processing and Applied Mathematics: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009, Revised Selected Papers, Part II 8.* Springer, 270–279.

[9] Luigi Luca Cavalli-Sforza, Italo Barrai, and Anthony WF Edwards. 1964. Analysis of human evolution under random genetic drift. In *Cold Spring Harbor symposia on quantitative biology*, Vol. 29. Cold Spring Harbor Laboratory Press, 9–20.

[10] Wentao Chen, Hailong Qiu, Jian Zhuang, Chutong Zhang, Yu Hu, Qing Lu, Tianchen Wang, Yiyu Shi, Meiping Huang, and Xiaowe Xu. 2021. Quantization of deep neural networks for accurate edge computing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 17, 4 (2021), 1–11.

[11] Gillian Chu and Tandy Warnow. 2023. SCAMPP+ FastTree: improving scalability for likelihood-based phylogenetic placement. *Bioinformatics Advances* 3, 1 (2023), vbad008.

[12] Reinout Corts and Nikolaos Alachiotis. 2023. A Survey of Processing Systems for Phylogenetics and Population Genetics. *ACM Transactions on Reconfigurable Technology and Systems* 16, 3 (2023), 1–27.

[13] ds950 [n. d.]. *Versal Architecture and Product Data Sheet.* https://docs.amd.com/v/u/en-US/ds950-versal-overview

[14] ds957 [n. d.]. *Versal AI Core Series Data Sheet: DC and AC Switching Characteristics.* https://docs.amd.com/r/en-US/ds957-versal-ai-core/Summary

[15] Joseph Felsenstein. 1978. Cases in which parsimony or compatibility methods will be positively misleading. *Systematic zoology* 27, 4 (1978), 401–410.

[16] Joseph Felsenstein. 1981. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of molecular evolution* 17 (1981), 368–376.

[17] Tomas Flouri, F Izquierdo-Carrasco, Diego Darriba, Andre J Aberer, L-T Nguyen, BQ Minh, Arndt Von Haeseler, and Alexandros Stamatakis. 2015. The phylogenetic likelihood library. *Systematic biology* 64, 2 (2015), 356–362.

[18] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: VersalTM architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 84–93. https://doi.org/10.1145/3289602.3293906

[19] Brandon S Gaut, Spencer V Muse, W Dennis Clark, and Michael T Clegg. 1992. Relative rates of nucleotide substitution at the rbc L locus of monocotyledonous plants. *Journal of molecular evolution* 35 (1992), 292–303.

[20] Calvin LC Goemann, Royce Wilkinson, William Henriques, Huyen Bui, Hannah M Goemann, Ross P Carlson, Sridhar Viamajala, Robin Gerlach, and Blake Wiedenheft. 2023. Genome sequence, phylogenetic analysis, and structure-based annotation reveal metabolic potential of Chlorella sp. SLA-04. *Algal Research* 69 (2023), 102943.

[21] Diep Thi Hoang, Olga Chernomor, Arndt Von Haeseler, Bui Quang Minh, and Le Sy Vinh. 2018. UFBoot2: improving the ultrafast bootstrap approximation. *Molecular biology and evolution* 35, 2 (2018), 518–522.

[22] Konstantin Hoffmann, Remco Bouckaert, Simon J Greenhill, and Denise Kühnert. 2021. Bayesian phylogenetic analysis of linguistic data using BEAST. *Journal of Language Evolution* 6, 2 (2021), 119–135.

[23] Wim Hordijk and Olivier Gascuel. 2005. Improving the efficiency of SPR moves in phylogenetic tree search methods based on maximum likelihood. *Bioinformatics* 21, 24 (2005), 4338–4347.

[24] Taishan Hu, Nilesh Chitnis, Dimitri Monos, and Anh Dinh. 2021. Next-generation sequencing technologies: An overview. *Human Immunology* 82, 11 (2021), 801–811.

[25] Fernando Izquierdo-Carrasco, Nikolaos Alachiotis, Simon Berger, Tomas Flouri, Solon P Pissis, and Alexandros Stamatakis. 2013. A generic vectorization scheme and a GPU kernel for the phylogenetic likelihood library. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 530–538.

[26] Paschalia Kapli, Ziheng Yang, and Maximilian J Telford. 2020. Phylogenetic tree building in the genomic age. *Nature Reviews Genetics* 21, 7 (2020), 428–444.

[27] Yuanning Li, Jacob L Steenwyk, Ying Chang, Yan Wang, Timothy Y James, Jason E Stajich, Joseph W Spatafora, Marizeth Groenewald, Casey W Dunn, Chris Todd Hittinger, et al. 2021. A genome-scale phylogeny of the kingdom Fungi. *Current Biology* 31, 8 (2021), 1653–1665.

[28] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* 461 (2021), 370–403.

[29] Xingchao Liu, Mao Ye, Dengyong Zhou, and Qiang Liu. 2021. Post-training quantization with multiple points: Mixed precision without mixed precision. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 8697–8705.

[30] Alessandra Lo Presti, Federica Del Chierico, Annamaria Altomare, Francesca Zorzi, Giovanni Monteleone, Lorenza Putignani, Silvia Angeletti, Michele Cicala, Michele Pier Luca Guarino, and Massimo Ciccozzi. 2023. Phylogenetic analysis of Prevotella copri from fecal and mucosal microbiota of IBS and IBD patients. *Therapeutic Advances in Gastroenterology* 16 (2023), 17562848221136328.

[31] Jinzhao Long, Juna Geng, Yake Xu, Yuefei Jin, Haiyan Yang, Yuanlin Xi, Shuaiyin Chen, and Guangcai Duan. 2022. Large-scale phylogenetic analysis reveals a new genetic clade among Escherichia coli O26 strains. *Microbiology Spectrum* 10, 1 (2022), e02525–21.

[32] Pavlos Malakonakis, Andreas Brokalakis, Nikolaos Alachiotis, Evripides Sotiriades, and Apostolos Dollas. 2020. Exploring modern FPGA platforms for faster phylogeny reconstruction with RAxML. In *2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (BIBE)*. IEEE, 97–104.

[33] Luke W Meredith, William L Hamilton, Ben Warne, Charlotte J Houldcroft, Myra Hosmillo, Aminu S Jahun, Martin D Curran, Surendra Parmar, Laura G Caller, Sarah L Caddy, et al. 2020. Rapid implementation of SARS-CoV-2 sequencing to investigate cases of health-care associated COVID-19: a prospective genomic surveillance study. *The Lancet infectious diseases* 20, 11 (2020), 1263–1271.

[34] Yu K Mo, Matthew W Hahn, and Megan L Smith. 2024. Applications of machine learning in phylogenetics. *Molecular Phylogenetics and Evolution* 196 (2024), 108066.

[35] Luca Nesterenko, Bastien Boussau, and Laurent Jacob. 2022. Phyloformer: towards fast and accurate phylogeny estimation with self-attention networks. *bioRxiv* (2022), 2022–06.

[36] Lam-Tung Nguyen, Heiko A Schmidt, Arndt Von Haeseler, and Bui Quang Minh. 2015. IQ-TREE: a fast and effective stochastic algorithm for estimating maximum-likelihood phylogenies. *Molecular biology and evolution* 32, 1 (2015), 268–274.

[37] Sang-Cheol Park, Kihyun Lee, Yeong Ouk Kim, Sungho Won, and Jongsik Chun. 2019. Large-scale genomics reveals the genetic characteristics of seven species and importance of phylogenetic distance for estimating pan-genome size. *Frontiers in microbiology* 10 (2019), 834.

[38] Jody Phelan, Wouter Deelder, Daniel Ward, Susana Campino, Martin L Hibberd, and Taane G Clark. 2020. Controlling the SARS-CoV-2 outbreak, insights from large scale whole genome sequences generated across the world. *BioRxiv* (2020), 2020–04.

[39] César Piñeiro, José M Abuín, and Juan C Pichel. 2020. Very Fast Tree: speeding up the estimation of phylogenies for large alignments through parallelization and vectorization strategies. *Bioinformatics* 36, 17 (2020), 4658–4659.

[40] Frederico Pratas, Pedro Trancoso, Alexandros Stamatakis, and Leonel Sousa. 2009. Fine-grain parallelism using multi-core, cell/BE, and GPU systems: accelerating the phylogenetic likelihood function. In *2009 International Conference on Parallel Processing*. IEEE, 9–17.

[41] Kelsey D Pugh. 2022. Phylogenetic analysis of Middle-Late Miocene apes. *Journal of Human Evolution* 165 (2022), 103140.

[42] Bruce Rannala and Ziheng Yang. 1996. Probability distribution of molecular evolutionary trees: a new method of phylogenetic inference. *Journal of molecular evolution* 43 (1996), 304–311.

[43] Mingming Ren, Xiaomin Huang, Zhuofeng Wu, Rebecca J Stones, Xiaoguang Liu, and Gang Wang. 2020. CuCodeML: GPU-Accelerated CodeML for the Branch-Site Model in Phylogenetics. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 67–75.

[44] FJLOJ Rodriguez, Jose L Oliver, Antonio Marín, and Juan R Medina. 1990. The general stochastic model of nucleotide substitution. *Journal of theoretical biology* 142, 4 (1990), 485–501.

[45] Fredrik Ronquist, Maxim Teslenko, Paul Van Der Mark, Daniel L Ayres, Aaron Darling, Sebastian Höhna, Bret Larget, Liang Liu, Marc A Suchard, and John P Huelsenbeck. 2012. MrBayes 3.2: efficient Bayesian phylogenetic inference and model choice across a large model space. *Systematic biology* 61, 3 (2012), 539–542.

[46] Sergio Santander-Jiménez, Aleksandar Ilic, Leonel Sousa, and Miguel A Vega-Rodríguez. 2017. Accelerating the phylogenetic parsimony function on heterogeneous systems. *Concurrency and Computation: Practice and Experience* 29, 8 (2017), e4046.

[47] seq-gen [n. d.]. *Sequence-Generator: An application for the Monte Carlo simulation of molecular sequence evolution along phylogenetic trees. Version 1.3.2.* https://snoweye.github.io/phyclust/document/Seq-Gen.v.1.3.2/Seq-Gen.Manual.html

[48] Megan L Smith and Matthew W Hahn. 2023. Phylogenetic inference using generative adversarial networks. *Bioinformatics* 39, 9 (2023), btad543.

[49] Stephen A Smith, Jeremy M Beaulieu, and Michael J Donoghue. 2009. Mega-phylogeny approach for comparative biology: an alternative to supertree and supermatrix approaches. *BMC evolutionary biology* 9 (2009), 1–12.

[50] Alexandros Stamatakis. 2014. RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics* 30, 9 (2014), 1312–1313.

[51] Edward Susko and Andrew J Roger. 2021. Long branch attraction biases in phylogenetics. *Systematic Biology* 70, 4 (2021), 838–843.

[52] Xudong Tang, Leonardo Zepeda-Nuñez, Shengwen Yang, Zelin Zhao, and Claudia Solís-Lemus. 2024. Novel symmetry-preserving neural network model for phylogenetic inference. *Bioinformatics Advances* 4, 1 (2024), vbae022.

[53] Cuong Than and Luay Nakhleh. 2008. SPR-based tree reconciliation: Non-binary trees and multiple solutions. In *Proceedings of the 6th Asia-Pacific Bioinformatics Conference*. World Scientific, 251–260.

[54] ug1079 [n. d.]. *AI Engine Kernel and Graph Programming Guide.* https://docs.amd.com/r/2022.2-English/ug1079-ai-engine-kernel-coding/Graph-Programming-Model

[55] Nicole L Washington, Karthik Gangavarapu, Mark Zeller, Alexandre Bolze, Elizabeth T Cirulli, Kelly M Schiabor Barrett, Brendan B Larsen, Catelyn Anderson, Simon White, Tyler Cassens, et al. 2021. Emergence and rapid transmission of SARS-CoV-2 B. 1.1. 7 in the United States. *Cell* 184, 10 (2021), 2587–2594.

[56] Victor Wijhe. 2024. *Signal Processing with AMD Adaptive Compute Acceleration Platform (ACAP) for Applications in Radio Astronomy.* Master's thesis. University of Twente.

[57] Steven Wijnja and Nikolaos Alachiotis. 2024. SoftCache: A Software Cache for PCIe-Attached Hardware Accelerators. In *Proceedings of the Platform for Advanced Scientific Computing Conference.* 1–11.

[58] Youwei Xiao and Yun Liang. 2023. Flexible Acceleration Framework for Dense/Sparse Matrix Multiplication on Versal ACAP. In *2023 International Symposium of Electronics Design Automation (ISEDA).* IEEE, 01–02.

[59] xilinx-best-practices [n. d.]. *Xilinx AI Engine Kernel Coding Best practices guide.* https://xilinx.eetrend.com/files/2021-11/wen_zhang_/100555327-227260-ug1079-ai-engine-kernel-coding.pdf

[60] Ziheng Yang. 1994. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: approximate methods. *Journal of Molecular evolution* 39 (1994), 306–314.

[61] Ziheng Yang. 1994. Statistical properties of the maximum likelihood method of phylogenetic estimation and comparison with distance matrix methods. *Systematic biology* 43, 3 (1994), 329–342.

[62] Ziheng Yang. 1996. Among-site rate variation and its impact on phylogenetic analyses. *Trends in ecology & evolution* 11, 9 (1996), 367–372.

[63] Ziheng Yang. 1996. Phylogenetic analysis using parsimony and likelihood methods. *Journal of Molecular Evolution* 42 (1996), 294–307.

[64] Ziheng Yang and Bruce Rannala. 2012. Molecular phylogenetics: principles and practice. *Nature reviews genetics* 13, 5 (2012), 303–314.

[65] Zhuoping Yang, Jinming Zhuang, Jiaqi Yin, Cunxi Yu, Alex K Jones, and Peipei Zhou. 2023. AIM: Accelerating Arbitrary-precision Integer Multiplication on Heterogeneous Reconfigurable Computing Platform Versal ACAP. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD).* IEEE, 1–9.

[66] Tjalling J Ypma. 1995. Historical development of the Newton–Raphson method. *SIAM review* 37, 4 (1995), 531–551.

[67] Yongqing Zhang, Qiang Zhang, Jiliu Zhou, and Quan Zou. 2022. A survey on the algorithm and development of multiple sequence alignment. *Briefings in bioinformatics* 23, 3 (2022), bbac069.

[68] Jianfu Zhou, Xiaoguang Liu, Douglas S Stones, Qiang Xie, and Gang Wang. 2011. MrBayes on a graphics processing unit. *Bioinformatics* 27, 9 (2011), 1255–1261.

[69] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, et al. 2023. CHARM: C omposing H eterogeneous A ccele R ators for M atrix Multiply on Versal ACAP Architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* 153–164.

[70] Jinming Zhuang, Zhuoping Yang, and Peipei Zhou. 2023. High performance, low power matrix multiply design on acap: from architecture, design challenges and dse perspectives. In *2023 60th ACM/IEEE Design Automation Conference (DAC).* IEEE, 1–6.

[71] Stephanie Zierke and Jason D Bakos. 2010. FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. *BMC bioinformatics* 11, 1 (2010), 1–12.

[72] Zhengting Zou, Hongjiu Zhang, Yuanfang Guan, and Jianzhi Zhang. 2020. Deep residual neural networks resolve quartet molecular phylogenies. *Molecular biology and evolution* 37, 5 (2020), 1495–1507.

[73] Marketa Zvelebil and Jeremy O Baum. 2007. *Understanding bioinformatics.* Garland Science.

Received October 8, 2024