A generic Vectorization Scheme and a GPU kernel for the Phylogenetic Likelihood Library

Fernando Izquierdo-Carrasco[1]*, Nikolaos Alachiotis[2]*, Simon Berger[2]*,
Tomáš Flouri[3]*, Solon P. Pissis[4]*† and Alexandros Stamatakis*
**The Exelixis Lab*
Scientific Computing Group
Heidelberg Institute for Theoretical Studies
Heidelberg, Germany
Email: {fernando.izquierdo, nikolaos.alachiotis, simon.berger, tomas.flouri, alexandros.stamatakis}@h-its.org
†Florida Museum of Natural History
University of Florida
Gainesville, Florida, USA
Email: spissis@flmnh.ufl.edu

Abstract—**Highly optimized library implementations for important scientific kernels can improve scientific productivity. To this end, we are currently developing the Phylogenetic Likelihood Library (PLL) that implements functions to compute and optimize the phylogenetic likelihood score on evolutionary trees. Here, we focus on novel techniques to orchestrate likelihood computations on large vector-like processors such as GPUs. We present a novel scheme for vectorizing computations and organizing conditional likelihood arrays (CLAs) in such a way that they do not need to be transferred at all between the GPU and the CPU. We compare the performance of our GPU implementation for DNA data with a highly optimized x86 version of the PLL that relies on manually tuned AVX intrinsics. Our GPU implementation accelerates the likelihood computations by a factor of two compared to the, most probably, currently fastest available x86 implementation. We conclude that, a hybrid GPU-CPU version needs to be developed and integrated into the PLL to leverage the computational power of modern desktop systems and clusters.**

Keywords-**phylogenetics; maximum likelihood; GPU; OpenCL; vector intrinsics;**

I. Introduction

The design and implementation of reusable software components as part of software libraries contributed substantially to the rapid development and deployment of software tools in the field of bioinformatics over the past years.

One of the advantages of such libraries is the amount of man-power that can be saved by not re-inventing the wheel. Moreover, library developers typically employ unit testing as part of their development cycle. Therefore, in combination with bug reports from the community that uses the library, the amount of bugs may be kept comparatively low. However, the major advantage of using software libraries is that they often give the user access to parallelized and hardware-based optimized function kernel implementations. These functions may not only significantly improve the performance of the tool that is being developed, but also saves countless and unnecessary man-hours that are wasted to efficiently re-implement a fundamental function.

The phylogenetic likelihood function (PLF) introduced by Joe Felsenstein in 1981 [1] is one of the most important statistical functions in the area of evolutionary Bioinformatics. It is *the* fundamental computational kernel for a plethora of widely-used and highly cited Maximum Likelihood and Bayesian tree inference programs such as MrBayes [2], PHYML [3], RAxML [4], GARLI [5], etc. Moreover, it is also essential for Bayesian programs that infer divergence times using the relaxed molecular clock model such as BEAST [6] or DPPDiv [7]. In all of the above applications, the phylogenetic likelihood function (PLF) dominates both, inference times (typically accounting for 80 - 95% of total execution time) and overall memory requirements (accounting for at least 70% of total RAM consumption). To this end, numerous efforts have been undertaken to optimize, parallelize, and accelerate the PLF as well as to reduce its memory requirements [8], [9]. This is required to (i) build competitive tools and (ii) keep up with the molecular data deluge.

Provided over ten years of experience in optimizing and parallelizing the PLF, we thus decided, to develop a Phylogenetic Likelihood Library (PLL) that incorporates all computational techniques that have been developed during the past years. Ideally, the PLL will allow researchers to focus on model and algorithm development without having to re-invent the wheel by developing their own, *ad hoc* PLF and possibly inefficient implementation.

[1]Supported by DFG project STA-860/3
[2]Supported by DFG project STA-860/2
[3]Supported by DFG project STA-860/4
[4]This work was supported in part by the NSF-funded iPlant Collaborative (NSF grant #DBI-0735191)

The PLL is based on the highly tuned PLF implementation of RAxML [4] and supports SSE3 and AVX intrinsics, as well as fine-grain PThreads and MPI parallelizations that rely on a master-worker scheme for executing parallel regions. The PLL has already been successfully deployed to substantially accelerate DPPDIV [7], a Bayesian program for divergence time(s) estimates.

Here, we focus on the development of a new, generic vectorization scheme for the PLF that allows to transparently deploy vector units of arbitrary length for PLF computations. We consider x86 intrinsics (128-bit wide SSE3 instructions and 256-bit wide AVX instructions) as well as GPUs as target vector units. A generic vectorization scheme is important to ensure portability of the code to increasing vector lengths (e.g., the 512-bit wide vector units of the new Xeon Phi processor). We also introduce a GPU memory organization scheme that reduces the amount of data that needs to be transferred between the GPU and the CPU to an absolute minimum, thereby improving performance. Furthermore, we provide a direct GPU implementation of the Newton-Raphson optimization procedure that is required to optimize branch lengths in Maximum Likelihood implementations.

Our long-term goals are to provide a highly optimized open-source library that entails state-of-the art implementations for all common input datatypes (DNA, protein data, etc) and that can be executed transparently on a large number of emerging parallel architectures. Hence, while we focus on technical issues here, one of the key future software engineering challenges will consist in designing an easy-to-use API.

According to our experiments our GPU implementation of the PLF is approximately twice as fast as the highly tuned x86 version of the PLF that relies on manually inserted and optimized AVX vector intrinsics.

The remainder of this paper is organized as follows: In Section II we survey related work on PLF library and GPU implementations. In Section III we provide an overview over the library. Thereafter, we describe the generic vectorization scheme in Section IV. In the subsequent Section V we cover technical details of the GPU implementation. Thereafter, we describe the experimental setup and the results we obtained (Section VI) and conclude in Section VII.

II. Related work

Early work on porting the RAxML likelihood functions to GPUs in the pre-CUDA and pre-OpenCL era was reported in [10].

Exploiting fine-grain parallelism with GPUs for the PLF has previously been addressed in [11] and [12] for MrBayes [2]. However, these implementations represent case studies or only cover a small subset (for specific data types such as DNA data) of the PLF in MrBayes. Hence, these initial efforts do not represent production-level implementations, but rather proof-of-concept studies.

The BEAGLE [13] (general purpose library for evaluating the likelihood of sequence evolution on trees) library introduced an application programming interface (API) for PLF computations and also offers efficient implementations thereof. BEAGLE can exploit modern hardware using SSE3 intrinsics, multi-threading, and GPUs. It has been integrated into Bayesian programs (BEAST [14] and MrBayes [2]) and Maximum Likelihood programs (GARLI [5]). The BEAGLE paper [13] reports performance results for DNA and Codon data on two 15-taxon datasets. The test datasets contained 8558 unique nucleotide (DNA) site patterns and 6080 unique codon site patterns respectively. For each of the three programs integrated with BEAGLE, the authors measured the speedup of the BEAGLE CPU, SSE3, and GPU (under single and double precision) implementations with respect to the corresponding native implementations. The largest speedups were obtained by the GPU implementation. For GARLI, only GPU speedups were reported (factor 3.8 for DNA data and 12 for codon data under double precision). The BEAGLE-based version of MrBayes yielded a maximum speedup of 16 (DNA data) and of 31 (Codon data) on the GPU using double precision arithmetics. Note that the relative speedup for MrBayes comparing the BEAGLE CUDA against the BEAGLE SSE3 performance was approximately 4.6 for DNA data. BEAST showed similar speedups for the GPU implementation under double precision (14-fold for DNA data and 37-fold for codon data). The speedups for single precision were larger. However, for large-scale real-world datasets (in particular with a high number of taxa), double precision arithmetics are typically required to guarantee numerical stability of the PLF [15].

Our PLL offers additional features that BEAGLE does not support. The PLL can also use AVX intrinsics. Furthermore, it implements numerical optimization functions such as the Newton-Raphson method for branch length optimization. BEAGLE defers these tedious programming tasks to the application programmer. It only offers functions for computing the first and second derivative of the likelihood function that can then be used by the application programmer to implement a Newton-Raphson branch length optimization procedure. Moreover, BEAGLE does currently not allow for conducting partitioned analyses which, given that partitioned analyses (distinct sets of likelihood model parameters are estimated for different parts of the multiple sequence alignment) are becoming increasingly common, represents a drawback of BEAGLE. As a consequence, BEAGLE does also not implement techniques [16], [17] for improving parallel load balance for partitioned analyses. Unlike the PLL, it does not offer a fine-grain MPI parallelization of the PLF and is hence limited to stand-alone shared memory nodes. Finally, BEAGLE does not implement the CAT model of rate heterogeneity [18] which can yield substantial computational savings in terms of floating point operations *and* memory compared to the standard Γ model of rate heterogeneity [19].

III. Phylogenetic Likelihood Library

The *Phylogenetic Likelihood Library* (PLL) is a parallelized and highly optimized software library derived from RAxML [4], a software tool for inference of large phylogenies under maximum likelihood. The current PLL code comprises implementations of state-of-the-art algorithms and data structures along with low-level technical and hardware-dependent optimizations. The library can calculate (and optimize) the likelihood on a phylogenetic tree for a plethora of statistical models and data types. It also implements branch length optimization and various tree alteration mechanisms, such as *Subtree Pruning and Regrafting* (SPR) and *Nearest Neighbor Interchanges* (NNI). Moreover, PLL can use multiprocessor architectures via a fine-grain parallelization of the PLF that relies on PThreads or MPI. In the parallel version, the alignment sites (or alignment partitions) are distributed across processors. Single x86 cores use SSE3 or AVX intrinsics to accelerate computations.

In the following, we describe in more detail those PLL functions which are required to outline the vectorization scheme and GPU implementation later-on.

For computing the likelihood on a tree, we need the following two core functions:

- The `newview()` function updates a conditional likelihood vector given two child nodes and given two transition probability matrices P for the respective branch lengths leading to these child nodes.
- The `evaluate()` function is called at the virtual root that has been placed into the unrooted tree for scoring it. Given the two conditional likelihood arrays at either end of the rooted branch and the branch length, `evaluate()` computes the overall log likelihood of the tree.

Usually, to compute the log likelihood of a tree, we need to conduct a post-order traversal of the tree topology (starting at the virtual root) and invoke `newview()` at each inner node. Once our post-order traversal reaches the virtual root, we invoke `evaluate()` to calculate the overall log likelihood score. Note that, `evaluate()` and `newview()` are sufficient to implement a Bayesian inference program, since the MCMC procedure, unlike the maximum likelihood method, does not require dedicated parameter optimization routines.

In the Maximum Likelihood (ML) framework however, we do need such explicit parameter optimization routines. Typically, branch length optimization is implemented via the Newton-Raphson procedure. Also note that, branch length optimization accounts for approximately 30% of total execution time in state-of-the-art ML tree inference algorithms. To optimize a specific branch, we need to invoke `newview()` first on the nodes at either end of the branch to ensure that the conditional likelihood arrays (CLAs; frequently also called ancestral probability vectors) are consistent with the branch that is being optimized. In fact, this corresponds to placing the virtual root of the tree into the branch that shall be optimized. Furthermore, we also need to invoke `newview()` when a branch has been changed to ensure that the conditional likelihood arrays in the tree are in a consistent state and reflect the changed branch.

The PLL implements the following two routines for the Newton-Raphson branch length optimization procedure:

- `coreDerivative()` computes the first and second derivative of the likelihood function at a given branch.
- `sumGAMMA()` pre-computes the element-wise product of the CLAs to the left and the right of the branch under optimization. This product is then re-used repeatedly by iterations of `coreDerivative()` and allows to save time by avoiding redundant computations.

Also note that, the PLL provides a function for direct branch length optimization (called `makenewz()`) using the Newton-Raphson procedure that uses the two functions described above.

IV. Generic Vectorization

An important part of the PLF is the `newview()` function that computes the CLAs at inner nodes of the tree in the course of a post-order traversal. The entries of these CLAs are computed according to Equation 1. The innermost loop of `newview()` calculates the sum over products between elements of the transition probability matrix P and corresponding elements in the CLA L.

$$\begin{aligned} L_i(x_i) = &\left[\sum_{x_j=A}^{T} P_{x_i x_j}(b_j) L_j(x_j)\right] \\ &\times \left[\sum_{x_k=A}^{T} P_{x_i x_k}(b_j) L_k(x_k)\right] \end{aligned} \tag{1}$$

This Equation is equivalent to calculating the scalar product between rows of P and *regions* (sub-vectors) of L. The number of matrix columns and hence the length of the sub-vectors corresponds to the number of states in the model and data. For DNA data there are four states (`A`, `C`, `G` and `T`) and for protein data there are 20 states. For the sake of simplicity, let us consider DNA data. It is important to note that, depending on the model of nucleotide substitution, there can often be more than one probability value per character/state at each site/element of the CLA. The widely-used Γ model of among-site rate heterogeneity [19], integrates the likelihood over the Γ function. This integral is approximated by discretizing the Γ function via usually four discrete rates (to save computations and memory). Hence, likelihood computations under the Γ model with four discrete rates require calculating and storing 16 values per alignment site in a CLA element (four values for `A`, four values for `C`, etc.).

site 0		site 1		site 2		site 3	
rate 0	rate 1	rate 0	rate 1	rate 0	rate 1	rate 0	rate 1
A C G T	A C G T	A C G T	A C G T	A C G T	A C G T	A C G T	A C G T

Figure 1. Memory layout of a CLA with a vector width of 1 ($VW := 1$)

site 0/1 interleaved		site 2/3 interleaved	
rate 0	rate 1	rate 0	rate 1
AA CC GG TT	AA CC GG TT	AA CC GG TT	AA CC GG TT

Figure 2. Memory layout of a CLA with a vector width of 2 ($VW := 2$)

To simplify the following illustrations we only use 2 discrete Γ rates however.

In a sequential implementation of `newview()`, the natural way to arrange the data in main memory is shown in Figure 1 (assuming DNA data with 4 states, 2 Γ rates, and 4 alignment sites). For each alignment site, the CLA contains 2 rate blocks. Each rate block contains 4 probabilities (1 per state, denoted by `A`, `C`, `G`, and `T`). Using this memory layout, the probability values of the states can be read efficiently from contiguous memory locations to calculate the scalar products in Equation 1.

This memory layout is used directly in the initial *ad hoc* SSE3 and AVX versions of RAxML. The calculation of the scalar products in the innermost `newview()` loop can be directly implemented by using element-wise multiply and horizontal add operations. However, this approach is only efficient, if the number of states (e.g., 4) is equal to or larger than the width of the vector unit. Since we use double precision floating point numbers this is the case both for SSE3 (vector width: 2), as well as AVX (vector width: 4) vector units. In contrast to this, modern GPUs have much wider vector units. In addition, the width of x86 vector units is also expected to increase (e.g., the Intel Xeon Phi). Hence, the initial *ad hoc* vectorization scheme can no longer be used for the PLL. Moreover, the manual vectorization for each model and data type combination proved to be error-prone and labor-intensive. Thus, we require a more generic vectorization scheme that is easier to port to new models and can conveniently be adapted to vector units of arbitrary length.

In order to use the wider vector units on GPUs, we introduce a new and more generic, vectorization scheme. Instead of exploiting parallelism within the innermost loop iteration of `newview()`, the new scheme now calculates a part of the conditional likelihood arrays simultaneously for multiple sites. We denote this approach as across-site vectorization. In principle, across-site vectorization is analogous to the sequential implementation of the PLF: The actual calculations of the scalar products in the innermost `newview()` loop are carried out sequentially. The main difference is that the scalar products are now being calculated for multiple sites (i.e., 2 or 4 sites for SSE3/AVX or more than 64 sites on the GPU) in parallel. In the SSE3 and AVX implementations this parallelism is exploited by using vector intrinsics. With SSE3 intrinsics, for instance, the `__m128` data type can be interpreted as a vector of 2x64bit double precision floating point values. It replaces the `double` data-type and the intrinsic vector operations `_mm_mul_pd` and `_mm_add_pd` are used instead of the `*` and + operators for 2x64bit vectors. Corresponding intrinsic functions exist for most of the built-in C/C++ data types and arithmetic operations, which yields the implementation of across-site vectorization relatively straight-forward. We have previously used a similar scheme for the inter-sequence vectorization of the PaPaRa 2.0 [20] dynamic programming algorithm. This simple vectorization scheme can only be used when the data (i.e., the CLAs) are stored using an appropriate memory layout. Such a layout needs to allow for reading the probability values of a specific state (e.g., `A`) that belong to neighboring alignment sites from contiguous memory locations. Since this is not possible using the standard memory layout (see Figure 1), we introduce an appropriately adapted and flexible (regarding the vector unit width) memory layout.

To assess the efficiency of this more generic vectorization scheme, we implemented it on both CPUs (using SSE as well as AVX) *and* GPUs. The major change consists in an adapted memory layout for the CLAs which now allows to efficiently exploit across-site parallelism on CPUs and GPUs. Note that, SSE and AVX instructions currently do not offer efficient operations for loading data from non-contiguous sites (data locations) into vector registers (see Figure 1). Generally, GPUs offer greater flexibility with respect to loading values from non-contiguous memory locations (e.g., loading the values corresponding to state `A` and discrete Γ rate 0 of sites $0, 1, \ldots, 32$). However, to obtain 'good' GPU performance, values should be read from contiguous memory locations to coalesce read/write accesses and prevent bank conflicts. We have therefore generalized the CLA memory layout to store corresponding values from different sites in contiguous memory. This allows for accessing the data at contiguous memory locations for the vectorized version of Equation 1. The memory layout is parameterized by the desired vector unit width (VW). For $VW := 1$, the memory layout corresponds exactly to the original memory layout of RAxML (see Figure 1). The analogous layout for $VW := 2$ is shown in Figure 2. As outlined in these figures, corresponding values from different alignment sites (e.g., state `A`, discrete Γ rate 0 of sites 0 and 1) are located at contiguous memory locations. Therefore, they can directly be loaded into an SSE register via a single load operation. Given this altered and adaptive memory layout, implementing a vectorized version of Equation 1 for sites 0 and 1 becomes straight-forward. The scheme can, in principle, be extended to arbitrary vector widths (see Section V).

Note however, that there do exist some limitations. Equation 1 can only be vectorized for those sites that evolve according to the same model. In other words, the sites need to share the same P matrices and the same α shape parameter that determines the form of the Γ curve. Thus, if the dataset is partitioned (different parameters for different parts (e.g., genes) of the alignment are being estimated), the maximum vector width is limited by the number of sites that evolve according to the same model. Moreover, it is also difficult to apply the above scheme to the, otherwise very efficient, CAT model of rate heterogeneity [18]. Instead of integrating the likelihood over different rates, it assigns one rate category (out of typically 25) to each alignment site. This means that, there are at least 25 different P matrices and that Equation 1 can only be vectorized across those sites that evolve according to the *same* P matrix (rate category). Hence, devising a generic vectorization scheme for the CAT model, which nonetheless currently is only implemented in RAxML and FastTree 2.0 [21], still remains a challenge.

V. GPU IMPLEMENTATION

Our GPU implementation is based on two key performance considerations for GPUs.

The first design criterion is that severe performance penalties are induced by frequently transferring large amounts of data between the CPU and GPU. To this end, we devise a memory layout strategy that allows to update and store the CLAs (the main bulk of the memory used for PLF computations) exclusively on the GPU. We have developed an OpenCL kernel that implements this strategy. The host program on the CPU simply orchestrates the tree search and invoke PLF computations on the CLAs that reside on the GPU. More specifically, the host program only needs to pass the memory addresses, that is, the starting positions of a CLA (corresponding to a node of the tree) in GPU memory to the GPU. Apart from that, the CPU only needs to communicate the substantially smaller P matrices and one additional variable to the GPU. The variable indicates which PLF function (e.g., `newview()`, `evaluate()`, etc.) shall be executed. The kernel call then returns at most one or two floating-point values to the CPU. For instance, the overall log likelihood (when `evaluate()` is called), or the first and second derivative of the likelihood function (when `coreDerivative()` is invoked). When `newview()` is invoked, no values are returned because this function simply updates the CLAs that reside in GPU memory.

To conduct a realistic performance assessment for a real application using the PLL, we re-implemented the search algorithm of RAxML-Light [22] using our library. Note that, the PLL can, of course, be integrated with any likelihood-based software such as IQPNNI [23], GARLI [5], PHYML [3], or DPPDIV [7]. DPPDIV has already been integrated with PLL. We focus on the RAxML-Light search algorithm for three reasons: (i) the code has been developed in our lab and was hence easy to integrate, (ii) RAxML-Light implements an *ad hoc* SSE3 *and* AVX vectorization that can be compared with the performance of the generic vectorization scheme and (iii) RAxML-Light uses the currently probably most efficient likelihood implementation.

The second design criterion is associated with improved GPU thread performance when threads access contiguous memory positions in global memory. We address this by using the CLA memory layout presented in Section IV and adapting it to GPUs.

A. *Kernel Implementation*

Here we describe the GPU kernel that implements the PLF functions described in Section III and that account for more than 95% of total execution time in RAxML-Light.

In the GPU version of `newview()`, the three distinct cases: *tip-tip* (both children are leaves), *inner-tip* (one child is a leave and the other is an inner node), *inner-inner* (both children are inner nodes) have been merged into one generic case (*inner inner*). This also induces a change in the layout of the tip vectors which are stored in the form of inner CLAs, rather than as a look-up table that is indexed by the raw alignment sequence data (for details see [24]). While this doubles the memory requirements for storing CLAs, it simplifies the storage of vectors in GPU memory, as well as the OpenCL code implementation. Hence, we allocate space for storing $2 \times n - 2$ conditional likelihood arrays on the GPU, where n is the number of taxa in the multiple sequence input alignment.

Finally, the `newview()` function also implements a numerical scaling procedure to avoid numerical underflow in likelihood computations (for a detailed description, see [24]).

The GPU implementation of `evaluate()` was straightforward. Because of the changed CLA representation at the tips, we simply had to omit the case where the left or right node of the branch at which the likelihood is calculated is a tip.

With respect to branch length optimization, we observed that, on the GPU, the overhead for invoking the precomputation kernel (`sumGAMMA()`) and storing the results is larger than re-computing the product of the entries prior to each invocation of `coreDerivative()`. Hence, we merged `sumGAMMA()` and `coreDerivative()` into a single kernel.

Apart from the $2n - 2$ full CLAs, we also store the raw alignment sequence data as well as the so-called `tipVector` data structure on the GPU. The reason for this is that we need to re-calculate the conditional likelihood arrays at the tips each time we change the values in the instantaneous substitution matrix Q (e.g., when optimizing the parameters of the General Time Reversible model of nucleotide substitution in RAxML-Light), which is required to calculate P. Note that, changes to Q represent a frequent operation and are invoked thousands of times when the rates in the Q matrix are being optimized.

While the values in the tip arrays are normally expected to be constant, this is not the case for the numerical implementation of the PLF used in the PLL. In fact, the matrix of left Eigenvectors is multiplied with the tip probability vectors prior to any further likelihood calculations. This helps to save some computations later-on. Each time the values in the Q matrix are changed this induces a change of the Eigenvector decomposition. The Eigenvector decomposition is used to exponentiate Q for obtaining the transition probability matrix P for a given branch length t (i.e., $P(t) = e^{Qt}$). Thus, the CLAs at the tips need to be updated accordingly after changes to the Eigenvector decomposition. However, transferring n tip vectors from the CPU to the GPU had a deteriorating effect on performance. Thus, we chose to only transfer the substantially smaller `tipVector` variable that contains the product of the left Eigenvectors with all possible DNA states (this is the look-up table used in the non-GPU PLL implementation) from the CPU to the GPU. Once the `tipVector` has been received, the new conditional likelihood arrays can be computed efficiently on the GPU by using the `tipVector` and the raw alignment sequence data. The overhead of re-computing the CLAs at the tips is negligible; it accounts for less than 1% of total run-time. In contrast to this, transferring all n CLAs for the tips from the CPU to the GPU for each change in Q, induced a run time overhead of up to 70%.

B. *GPU Memory Organization*

In the following we list the data structures that must be stored in GPU memory to correctly calculate the likelihood:

- CLAs for the $n - 2$ inner nodes and n tips.
- The raw sequence alignment data of the tips that is required for re-assembling the conditional likelihood vectors at the tips when the Q matrix changes.
- The relatively small `tipVector` array that is also required for re-assembling the tip vectors (see above).
- The weight vector that indicates how many times an alignment site pattern occurs (this is also called site pattern compression and used in all standard PLF implementations).
- The `diagptable` array representing the P matrix/matrices for `newview()` and `evaluate()` invocations.
- A `globalScaler` array of size $2n-2$ for storing (and later-on un-doing) the number of scaling multiplications to avoid numerical underflow at each node of the tree.
- Buffers to accumulate results, sum the per-site log likelihoods, and sum over the number of scaling multiplications.

The memory requirements are dominated by the $2 \times n - 2$ CLAs. Under double precision, each array requires $sites \times gamma_{rates} \times 4 \times 8$ bytes. As mentioned before, the number of discrete Γ rates is usually set to $gamma_{rates} := 4$. This allows us to approximate the memory requirements in advance. The proof-of-concept implementation we present here assumes that enough memory is available on the GPU. When this is not the case, it is possible to apply memory reduction strategies as presented in [8] that trade the lack of memory by additional computations. Alternatively, the computations can be split among several GPUs.

C. *OpenCL Implementation*

OpenCL (Open Computing Language) is an open standard for parallel programming of heterogeneous systems. It

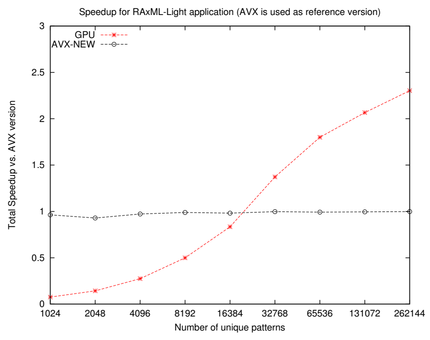

Figure 3. Speedups for a full application run of RAxML-Light. The reference is the AVX version with the standard layout.

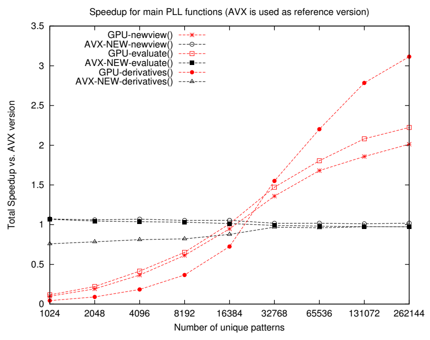

Figure 4. Speedups for each function of the PLL. The reference is the AVX version with the standard layout.

provides a language (a subset of ISO C99) for software developers to write portable code on SIMT (Single Instruction, Multiple Threads) architectures.

The OpenCL Execution Model consists of an application running on a Host (CPU), which offloads work to one or more Compute Devices (in our case a GPU). Each compute device is composed of one or more Compute Units. In CUDA (Compute Unified Device Architecture), these are called Streaming Multiprocessors (SM).

A Kernel represents the code for a work-item (thread), which are the basic units of work. Work-items are grouped into local work-groups (thread blocks). OpenCL applications can access various types of memory: Host memory (on the host CPU), global (visible to all workgroups, e.g. DRAM on the GPU board), local (shared within a workgroup), and private (registers per work-item). Global memory is accessed via 32-, 64-, or 128-byte transactions. To maximize global memory throughput, it is essential to optimize memory coalescence and minimize address scatter [25].

In OpenCL, the work-group size corresponds to the number of threads that are executed per streaming multiprocessor (SM). After experimenting with several multiples of 32, we empirically determined that a value of 64 worked best for our target application. This specific work-group size is optimal because, at each kernel call, all threads within a block can read data from contiguous positions in global memory. Thus, in our configuration we access 64 contiguous states that evolve according to the same discrete Γ rate category. Our GPU kernel initializes the tips, and reads/writes the CLAs according to this data layout ($VW := 64$).

In order to improve performance, we applied optimization techniques such as loop unrolling [25], and storing the transition probabilities matrices and the eigenvectors in shared memory (local memory for each SM). We also explicitly use registers to store global variables that are read and written several times during kernel execution.

VI. Experimental setup and results

We simulated DNA data sets of different dimensions using INDELible [26] on random trees under the Jukes-Cantor model. Initially, we used INDELible (v1.03) to generate a large alignment of 15 taxa (species) and 900,000 sites. We then used a ruby script to extract subsets of unique site patterns from this alignment such as to generate 15-taxon datasets with distinct numbers of unique sites. Thus, in these datasets, the number of sites actually corresponds to the number of distinct alignment patterns which facilitates the discussion of the results. Note that, using simulated data is fully sufficient for measuring the performance of the PLL GPU version.

We executed the default RAxML-Light [22] search algorithm (version 1.0.5) to infer trees on these datasets. We compared the GPU and AVX implementations (see Section V) that are based on the new memory layout (see Section IV) against the fastest serial version of RAxML-Light using the *ad hoc* AVX vectorization. The three code versions only differ with respect to their PLF implementations (`newview()`, `evaluate()`, `sumGAMMA()`, verb—coreDerivative()—). As mentioned before, these functions typically account for more than 95% of overall execution time and for 98% on the largest dataset with 262,144 unique site patterns. We manually instrumented the code to measure how much time is spent in each function. The three source codes and test datasets are available at http://www.exelixis-lab.org/gpu_pll_hicomb.tar.gz

It is straight-forward to verify code correctness since the RAxML-Light search algorithm is deterministic for given, fixed starting trees. Thus, it is sufficient to compare the resulting tree topologies and log likelihood scores.

To verify the correctness of the numerical scaling procedure, we also generated a dataset with 200 taxa (data not shown), to force the codes to conduct numerical scaling. Note that, the number of scaling multiplications is proportional to the number of taxa (see [24] for details).

We executed the AVX versions (standard and new layout) on an Intel i5-3550 CPU running at 3.30GHz with 8GB

Table I
TOTAL RUN TIMES (IN SECONDS) FOR THE RAxML-LIGHT SEARCH ALGORITHM

Patterns	1024	2048	4096	8192	16384	32768	65536	131072	262144
AVX	2.97	5.76	14.50	25.01	75.81	117.30	300.05	508.45	1503.48
AVX-NEW	3.09	6.20	14.90	25.29	77.23	117.57	302.24	510.82	1506.45
GPU	39.15	40.19	52.75	50.12	90.83	85.46	166.69	246.03	652.90

RAM. The GPU version was executed on the same host system, which is also equipped with a NVIDIA Tesla C2075 card (448 CUDA cores, 1.15GHz, and 6GB GDDR5 of memory).

The total runtimes are shown in Table I. We achieved overall speedups exceeding a factor of two for the longest test dataset (see Figure 3). The three PLL kernels show comparable speedups. We observed a maximum speedup of three for the derivative computation. For `newview()`, which consumes the largest fraction of execution time, we obtained a maximum speedup of 2.01 (see Figure 4).

Overall, the GPU speedups are rather mediocre and disappointing. This is mainly because we are comparing the GPU code to the probably fastest currently available PLF implementation that relies on code that has been manually vectorized and tuned for AVX intrinsics. Thus, the GPU speedups obtained for DNA data are substantially lower than those reported in the BEAGLE paper. However, the speedups reported for BEAGLE compare GPU performance of BEAST, MrBayes, and GARLI, to plain C and SSE3-based implementations. Nonetheless, the performance of our PLL GPU implementation is expected to improve when models/data with more states are used, because they perform more computations per data accesses than for DNA data. Note that, the amount of PLF floating point computations increases with the squared number of states.

Nonetheless, GPUs can yield a two-fold speedups over our highly optimized manual AVX implementation. Another interesting observation is that no performance penalty is induced by using the more generic vectorization scheme with AVX instructions.

In the final analysis, it becomes evident that, a hybrid CPU/GPU implementation making use of all available computational resources represents the best solution, despite some major challenges regarding load balance.

VII. CONCLUSION AND FUTURE WORK

We presented a GPU implementation for the main functions that are required for Bayesian and ML-based phylogenetic inference. This implementation is embedded into a larger project that aims at developing a portable and scalable phylogenetic likelihood library. In our approach, we store all conditional likelihood vectors in the GPU memory to avoid transferring large amounts of data between the GPU and the CPU. We have also introduced an alternative layout for the conditional likelihood vectors, which allows to exploit a larger number of threads in GPU computations via vector operations. It also facilitates porting the library to larger x86 vector units that will soon become available.

While the speedups we obtain may appear mediocre, one must keep in mind that they were obtained in comparison to the fastest currently available x86 implementation of these functions. In other words, we report as fair as possible speedups.

With respect to future work, we plan to fully integrate the GPU kernel with the PLL and support all models and data types (e.g., protein data and the CAT model of rate heterogeneity). We also intend to develop a hybrid implementation that can exploit all GPUs and x86 cores of a given system. We will also explore if the re-computation approach [8], that trades memory for additional computations can be applied to GPUs as well. Finally, we will also provide GPU support for analyzing partitioned datasets, which is already a feature of the x86 version of the PLL, and adapt our load-balancing strategies for this scenario.

REFERENCES

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

[2] F. Ronquist, M. Teslenko, P. van der Mark, D. L. Ayres, A. Darling, S. Hhna, B. Larget, L. Liu, M. A. Suchard, and J. P. Huelsenbeck, “Mrbayes 3.2: Efficient bayesian phylogenetic inference and model choice across a large model space,” *Systematic Biology*, 2012. [Online]. Available: http://sysbio.oxfordjournals.org/content/early/2012/03/02/sysbio.sys029.abstract

[3] S. Guindon, J. Dufayard, V. Lefort, M. Anisimova, W. Hordijk, and O. Gascuel, “New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of PhyML 3.0,” *Systematic biology*, vol. 59, no. 3, pp. 307–321, 2010.

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

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

[6] A. Drummond and A. Rambaut, “Beast: Bayesian evolutionary analysis by sampling trees,” *BMC evolutionary biology*, vol. 7, no. 1, p. 214, 2007.

[7] T. Heath, M. Holder, and J. Huelsenbeck, "A dirichlet process prior for estimating lineage-specific substitution rates," *Molecular biology and evolution*, vol. 29, no. 3, pp. 939–955, 2012.

[8] F. Izquierdo-Carrasco, J. Gagneur, and A. Stamatakis, "Trading running time for memory in phylogenetic likelihood computations," in *BIOINFORMATICS*, J. Schier, C. M. B. A. Correia, A. L. N. Fred, and H. Gamboa, Eds. SciTePress, 2012, pp. 86–95.

[9] F. Izquierdo-Carrasco, S. Smith, and A. Stamatakis, "Algorithms, data structures, and numerics for likelihood-based phylogenetic inference of huge trees," *BMC bioinformatics*, vol. 12, no. 1, p. 470, 2011.

[10] M. Charalambous, P. Trancoso, and A. Stamatakis, "Initial Experiences Porting a Bioinformatics Application to a Graphics Processor," in *Proc. of the 10th Panhellenic Conference on Informatics (PCI 2005)*, 2005, pp. 415–425.

[11] F. Pratas, P. Trancoso, L. Sousa, A. Stamatakis, G. Shi, and V. Kindratenko, "Fine-grain parallelism using multi-core, cell/be, and gpu systems," *Parallel Computing*, vol. 38, no. 8, pp. 365–390, 2012.

[12] J. Zhou, X. Liu, D. Stones, Q. Xie, and G. Wang, "Mrbayes on a graphics processing unit," *Bioinformatics*, vol. 27, no. 9, pp. 1255–1261, 2011.

[13] D. L. Ayres, A. Darling, D. J. Zwickl, P. Beerli, M. T. Holder, P. O. Lewis, J. P. Huelsenbeck, F. Ronquist, D. L. Swofford, M. P. Cummings, A. Rambaut, and M. A. Suchard, "BEAGLE: An Application Programming Interface and High-Performance Computing Library for Statistical Phylogenetics," *Systematic Biology*, vol. 61, no. 1, pp. 170–173, 2012.

[14] M. A. Suchard and A. Rambaut, "Many-core algorithms for statistical phylogenetics," *Bioinformatics*, vol. 25, no. 11, pp. 1370–1376, 2009.

[15] S. Berger and A. Stamatakis, "Accuracy and performance of single versus double precision arithmetics for maximum likelihood phylogeny reconstruction," *Parallel Processing and Applied Mathematics*, pp. 270–279, 2010.

[16] A. Stamatakis and M. Ott, "Load Balance in the Phylogenetic Likelihood Kernel," in *Proceedings of ICPP 2009*, 2009.

[17] J. Zhang and A. Stamatakis, "The multi-processor scheduling problem in phylogenetics," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 691–698.

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

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

[20] S. A. Berger and A. Stamatakis, "PaPaRa 2.0: A Vectorized Algorithm for Probabilistic Phylogeny-Aware Alignment Extension; Exelixis-RRDR-2012-5; http://sco.h-its.org/exelixis/pubs/Exelixis-RRDR-2012-5.pdf," Heidelberg Institute for Theoretical Studies, Tech. Rep., 2012. [Online]. Available: http://sco.h-its.org/exelixis/pubs/Exelixis-RRDR-2012-5.pdf

[21] M. Price, P. Dehal, and A. Arkin, "Fasttree 2–approximately maximum-likelihood trees for large alignments," *PLoS One*, vol. 5, no. 3, p. e9490, 2010.

[22] A. Stamatakis, A. J. Aberer, C. Goll, S. A. Smith, S. A. Berger, and F. Izquierdo-Carrasco, "RAxML-Light: a tool for computing terabyte phylogenies," *Bioinformatics*, vol. 28, no. 15, pp. 2064–2066, 2012.

[23] B. Minh, L. Vinh, A. Haeseler, and H. Schmidt, "pIQPNNI: parallel reconstruction of large maximum likelihood phylogenies," *Bioinformatics*, vol. 21, no. 19, pp. 3794–3796, 2005.

[24] A. Stamatakis, "Orchestrating the phylogenetic likelihood function on emerging parallel architectures," *Bioinformatics–High Performance Parallel Computer Architectures, B. Schmidt, Ed. CRC Press*, pp. 85–115, 2012.

[25] N. Alachiotis, S. Berger, and A. Stamatakis, "Coupling SIMD and SIMT Architectures to Boost Performance of a Phylogeny-aware Alignment Kernel," *BMC Bioinformatics*, vol. 13, no. 1, pp. 196+, 2012.

[26] W. Fletcher and Z. Yang, "INDELible: a flexible simulator of biological sequence evolution." *Molecular biology and evolution*, vol. 26, no. 8, pp. 1879–1888, 2009.