2017 46th International Conference on Parallel Processing Workshops

Heterogeneous Hardware Support in BEAGLE, a High-Performance Computing Library for Statistical Phylogenetics

Daniel L. Ayres and Michael P. Cummings
Center for Bioinformatics and Computational Biology
University of Maryland
College Park, MD, 20742, USA
ayres@umiacs.umd.edu and mike@umiacs.umd.edu

***Abstract*—We describe our approach to extend the BEAGLE library for high-performance statistical phylogenetic inference (maximum likelihood estimation and Bayesian analysis) in order to support a wider range of modern accelerators and multicore CPUs, and present the corresponding performance results from these platforms. Our solution includes a shared code design providing a uniform interface for a variety of compute platforms available under CUDA and OpenCL parallel computing frameworks. We have also implemented CPU threading in BEAGLE, and in sum these improvements allow the library to exploit a wide-range of hardware parallelism including CPU and Xeon Phi, vectorization intrinsics (e.g., SSE, AVX), and GPUs. Although code reuse and maintainability are features of our design, our approach also includes hardware-specific optimizations for performance critical portions of the code. Extending the BEAGLE library in this manner allows a greater variety of users to exploit the hardware resources available to them. As an example of increase in performance, using BEAGLE on a system with two Intel Xeon E5-2680v4 CPUs we observe a 39-fold speedup for a MrBayes 3.2.6 codon-model analysis, compared to the native MrBayes MPI-SSE implementation. The general design features of the library also provide a model for software development using parallel computing frameworks that is applicable to other domains.**

***Keywords*-Bayes methods; Biology computing; Evolution (biology); Phylogeny; Maximum likelihood estimation; Multicore processing; Parallel programming**

I. Introduction

Advances in computer hardware, specifically in parallel architectures, such as multicore CPUs, manycore CPUs (e.g., Intel Xeon Phi), GPUs, and CPU intrinsics (e.g., SSE, AVX), have created opportunities for new approaches to computationally intensive analysis methods. The design and development process required to take advantage of these parallel computing resources begins with decisions of what hardware to support and development frameworks to use. These initial decisions typically have implications that generally constrain applicable hardware used by the software developed, as well as the complexity of the software itself. Here we describe the software design and optimization approaches used to extend the range of hardware devices supported in the upcoming release of BEAGLE, a high-performance library for statistical phylogenetics [1]. We then explore the performance of the library on a variety of modern hardware resources and platforms.

Our design harnesses parallel hardware via multiple frameworks, and includes a model for sharing kernels for CUDA and OpenCL frameworks. Our own motivations for pursuing a development plan involving multiple frameworks comprise a number of elements. First among these is our desire to serve a large community of evolutionary biologists and others doing very common, but very computationally intensive calculations. This community has access to a broad range of hardware, and developing with multiple frameworks helps our library work across this range of hardware. Secondly, use of multiple frameworks diversifies risk across both hardware and software platforms. Market forces largely determine the composition of the hardware-software ecosystem, and correctly choosing the more important among the possible combinations can be difficult in the early phases of the hype cycle and in the presence of vendor marketing. Poor choice of target hardware or development framework can result in greatly diminished impact and a poorly served domain science community.

Regardless of high apparent promise of any particular option, at least initially, diversifying across processor architectures and development frameworks seems prudent. As examples of risk in the hardware realm consider the history of processors such as the Intel Itanium and STI Cell Broadband Engine, or the current status of the Intel Xeon Phi. In the realm of frameworks illustrative examples of risk include OpenCL for Apple macOS, for which not all features are supported, and OpenCL for Xeon Phi (Knights Landing), which is not available at the time of this writing. In addition to risk reduction, diversifying across processor architectures and development frameworks results in deeper understanding of hardware features and programming approaches, which can subsequently lead to better performance across implementations.

We continue this paper by providing an abbreviated review of related work, some context of the basic problem from the application domain science and computational perspectives, a general overview of the BEAGLE library, and follow with details regarding our shared framework strategy, including

1530-2016 2017

DOI 10.1109/ICPPW.2017.17

various design issues, hardware-specific optimizations, and performance results.

II. Related Work

We restrict our abbreviated consideration of related work involving to that involving the CUDA and OpenCL frameworks, as these comprise the most widely used for GPU programming, which is a special, though not exclusive, focus of the BEAGLE library. Furthermore, it is the users of the CUDA and OpenCL frameworks who are most likely to find some of our design decisions most applicable to their own efforts. This related work can be generally classified as translators, where the objective is to take code associated with one framework, most commonly CUDA for NVIDIA GPUs, and translate to another framework or processor architecture. These translators differ in the starting code for translation, as well as the target framework or processor architecture.

Starting with the pseudo-assembly code Parallel Thread Execution (PTX) generated in the CUDA framework, Ocelot [2] targets x86 and STI Cell Broadband Engine processors, whereas Caracal [3] targets the AMD Compute Abstraction Layer (CAL), a low-level access software development layer. Source-to-source translation from CUDA has been an approach followed by others. Among the direct source code translation approach are MCUDA [4] targeting x86-based processors, and CU2CL (CUDA-to-OpenCL) [5], [6], which targets OpenCL. Swan [7] also targets OpenCL, but requires the developer to replace CUDA API calls with intermediary equivalent calls in a Swan-specific syntax, and these, in turn, are translated to generate the OpenCL code.

Our own work described in this paper differs fundamentally from the work mentioned above. First, our approach is to design and develop kernels that are shared between CUDA and OpenCL, rather than to create or employ a tool for after-the-fact translation to a different framework or architecture. From our perspective and objectives this fundamental difference has several advantages over translation particularly in the areas of efficiency and simplicity: i) ready sharing of core algorithms; ii) easier accommodation of new analytical models; iii) no dependence on translators, which may or may not be up-to-date with respect to the latest framework versions, thus eliminating another potential development risk; iv) reduces duplicated code; and v) adroitly facilitates hardware-specific optimizations.

III. Evolutionary Biology, the Scientific Domain

Research in evolutionary biology can generally be divided as being most closely associated with either of two broad categories: i) *macroevolution*, which involves the processes of speciation and extinction; and ii) *microevolution*, which involves the processes affecting changes in the genetic structure of populations. These evolutionary categories converge in that trees representing ancestor-descendent relationships are central to the conceptual and analytical framework for both macro- and microevolution, which are embodied by phylogenetics and population genetics respectively.

In a broad sense phylogenetics is the study of evolutionary relationships. Typically, modern phylogenetic analyses involve obtaining DNA sequence data from a set of organisms, and using model-based methods to infer a binary tree. This tree represents the evolutionary history of the organisms going back to their most recent common ancestor and is, in essence, a subset of the overall tree of life.

A. Likelihood Function

The most effective methods for inferring both phylogenetic trees and gene genealogies are based on either maximum likelihood estimation or Bayesian analysis, which share the same computational bottleneck: calculation of the likelihood of trees [8]. When profiling GARLI [9], a leading phylogenetic inference program, we have observed that, for DNA models, likelihood related calculations typically constitute over 94% of the overall runtime. For more complex models (e.g., amino-acid or codon-based), likelihood calculation will typically incur an even greater proportion of the analysis time. Speeding the calculation of the likelihood function is key to increasing the performance of statistical inference-based phylogenetic analyses.

The core likelihood calculations apply to a subtree comprising a node (x_0) and its two descendant nodes (x_1 and x_2), and the connecting branches (of length t_1 and t_2), and is repeated for all such subtrees of the larger tree being considered. This partial-likelihoods function [8] is as follows.

$$L(x_0) = \left(\sum_{x_1} \Pr(x_1|x_0, t_1) L(x_1) \right) \times \left(\sum_{x_2} \Pr(x_2|x_0, t_2) L(x_2) \right) \quad (1)$$

This calculation is repeated for each site (i.e., sequence position), and for each possible character a site can assume (e.g., `a`, `c`, `g`, and `t`, for a nucleotide model sequence). The computational complexity of the likelihood calculation for a given tree is $O(p \times s^2 \times n)$, where p is the number of positions in the sequence (typically on the order of 10^2 to 10^6), s is the number of states each character in the sequence can assume (typically 4 for a DNA model, 20 for an amino-acid model, or 61 for a codon model), and n is the number of operational taxonomic units (e.g., species, alleles).

Thus, to explore even a fraction of the total search space, a very large number of topologies are evaluated, and hence a very great number of likelihood calculations have to be performed. This leads to analyses that can take days, weeks or even months to run. Further compounding the issue, rapid advances in the collection of DNA sequence data have made

the limitation for biological understanding of these data an increasingly computational problem.

The structure of the likelihood calculation, involving large numbers of positions and multiple states, as well as other characteristics, makes it a very appealing computational fit to modern parallel microarchitectures such as multi and manycore CPUs, and especially, GPUs.

IV. BEAGLE

BEAGLE [1] is a high-performance likelihood-calculation platform for phylogenetic applications. BEAGLE defines a uniform application programming interface (API) and includes a collection of efficient implementations for calculating a variety of phylogenetic models on different hardware devices, such as graphics processing units (GPUs), Intel Xeon Phi devices, and multicore CPUs.

The BEAGLE project has been very successful in bringing hardware accelerators to phylogenetics. The library was the first to focus on high-performance computation of the phylogenetic likelihood calculation via fine-scale parallelization. It is the most widely adopted library for this purpose and has been integrated into popular phylogenetics software including BEAST [10], MrBayes [11], and PhyML [12], and has been widely used for phylogenetic analyses. Recent work on the BEAGLE library identifying independent likelihood estimates in analyses of partitioned datasets and in proposed tree topologies, and configuring concurrent computation of these likelihoods via CUDA and OpenCL frameworks results in substantially increased performance [13].

Other proposals have been made to bring hardware acceleration to statistical phylogenetics, however these have typically focused only on MrBayes and have only applied to a subset of models the program supports [14], or have not made the source code or binaries available [15].

A. Overall Design

The general structure of the BEAGLE library version 1 can be conceptualized as layers (Fig. 1), the upper most of which is a C API. Alternatively, Java programs can use a Java Native Interface (JNI) wrapper, which is provided with the source code.

Underlying the API is an implementation management layer, which loads the available implementations, makes them available to the client program, and passes API commands to the selected implementation. Internally, the implementations in BEAGLE derive from two general models. One is a serial CPU implementation model that does not directly use external frameworks, and which comprises a standard CPU implementation, and one with added SSE intrinsics.

The other implementation model involves an explicit parallel accelerator programming model, which uses the CUDA external computing framework. This parallel implementation model communicates directly with the GPU via CUDA APIs.

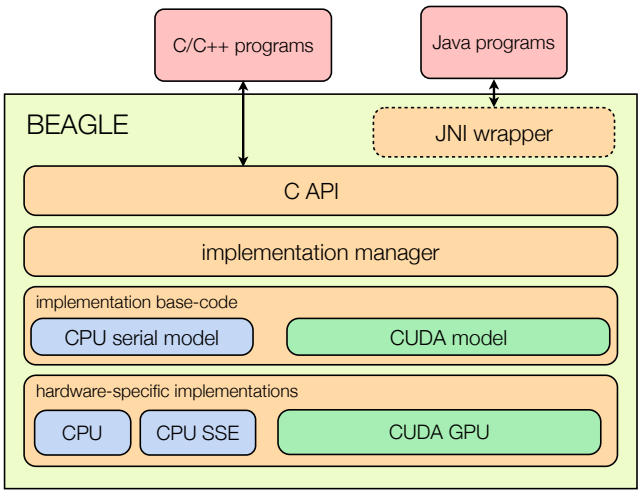

Figure 1. Layer diagram depicting the overall structure of the BEAGLE library version 1.

B. Application Programming Interface

The BEAGLE API was designed to increase performance via fine-scale parallelization while reducing data transfer and memory copy overhead to an external hardware accelerator device. To accomplish this, the library lacks the concept or data structure for a tree, which provides for a more simplified implementation in application programs. Instead, BEAGLE acts directly on flexibly indexed data storage which stores the partial-likelihoods.

C. Implementation Overview

The design of BEAGLE allows for new implementations to be developed without the need to alter the core library code or how client programs interface with the library. This architecture also includes a plugin system, which allows implementation-specific code (via shared libraries) to be loaded at runtime when the required dependencies are present. Consequently new frameworks and hardware platforms can more easily be made available to programs that use the library, and ultimately to users performing phylogenetic analyses.

D. CPU Implementations

BEAGLE version 1 includes a serial CPU implementation, as well as an SSE implementation for nucleotide models in double-precision, which uses vector processing extensions present in many CPUs to parallelize computation across character state values.

E. CUDA Implementation

The initial version of BEAGLE exclusively used the CUDA platform to exploit NVIDIA GPUs. It implemented novel computational methods for evaluating likelihoods under arbitrary molecular evolutionary models, harnessing the large

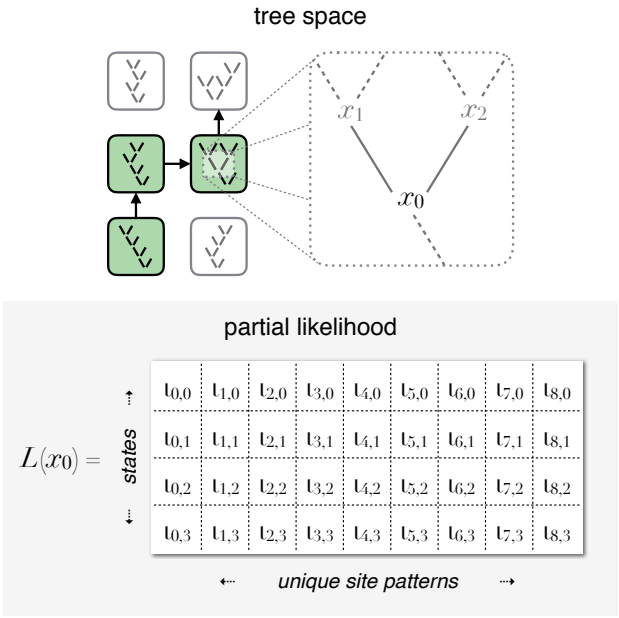

Figure 2. Diagrammatic example of the tree sampling process and fine-grained parallel computation of phylogenetic partial likelihoods using BEAGLE on GPUs for a nucleotide-model problem with 5 taxa and 9 site patterns. Each entry in a partial likelihood array L is assigned to a separate GPU thread t. In this simplified example, 45 GPU threads are created to enable parallel evaluation of each entry of the partial likelihood array $L(x_0)$ Calculations for real datasets would typically generate thousands of threads.

number of processing cores to efficiently parallelize calculations [1], [16]. We originally chose to develop with the CUDA Driver API rather than the Runtime API due to its greater flexibility. This also facilitated sharing code with the subsequently developed OpenCL solution.

F. Parallel Computation

BEAGLE exploits GPUs via fine-grained parallelization of functions necessary for computing the likelihood on a phylogenetic tree. Phylogenetic inference programs typically explore tree space in a sequential manner (Fig. 2, *tree space*) or with only a small number of sampling chains, offering limited opportunity for task-level parallelization. In contrast, the crucial computation of partial likelihood arrays at each node of a proposed tree presents an excellent opportunity for fine-grained data parallelism, for which GPUs are especially suited.

In order to calculate the overall likelihood of a proposed tree, phylogenetic inference programs perform a post-order traversal, evaluating a partial likelihood array at each node. When using BEAGLE, the evaluation of these multi-dimensional arrays is offloaded to the library. Though each partial likelihood array is still evaluated in series, BEAGLE assigns the calculation of the array entries to separate GPU threads, for computation in parallel (Fig. 2, *partial likelihood*). Further, BEAGLE uses GPUs to parallelize other functions necessary for computing the overall tree likelihood, thus minimizing data transfers between the CPU and GPU. These additional functions include those necessary for computing branch transition probabilities, for integrating root and edge likelihoods, and for summing site likelihoods.

Table I
SYSTEM SPECIFICATIONS

	system 1	*system 2*
CPU (Intel)	Core i7-930	Xeon E5-2680v4 (x2)
GPU 1 (NVIDIA)	Quadro P5000	—
GPU 2 (AMD)	Radeon R9 Nano	FirePro S9170
Linux kernel	4.8.13	3.10.0
GCC version	6.2.1	6.2.0
CUDA release	8.0	—
OpenCL driver 1	NVIDIA 375.26	Intel 1.2.0
OpenCL driver 2	AMD 1912.5	AMD 1800.8

For exploiting CPU parallelism, BEAGLE provides an SSE implementation that vectorizes likelihood calculations. Additionally, in order to exploit multiple CPU cores, application programs running partitioned analyses can invoke multiple library instances, one for each data subset (or partition). This approach suits the trend of increasingly large molecular sequence data sets, which are often heavily partitioned in order to better model the underlying evolutionary processes.

V. EXTENDING SUPPORTED HARDWARE IN BEAGLE

A. Benchmarking and Testing Methods

As we extended BEAGLE with new implementations, we further developed our test program (*genomictest*) to support a wider range of analysis types and more detailed output. This program generates random synthetic datasets of arbitrary sizes and is used to evaluate performance and assure correct functioning of the library.

For benchmarking we generate a measure of throughput in terms of the effective number of floating point operations per second for computation of the partial-likelihoods function (see equation III-A). In contrast to a direct timing benchmark, throughput allows us to more easily compare performance across different problem sizes and floating point precision formats. This measure also allows comparison to an upper performance bound and generally informs whether computations are compute or memory bound.

For assessing result correctness, we developed a set of testing scripts which evaluate different analyses types by varying input parameters to our *genomictest* program. These testing scripts are publicly available in the project repository and we have verified correct functioning of all new implementations described below.

Table I shows relevant hardware and software specifications for the two main systems used to perform the benchmarks results reported in this paper. Table II summarizes the

Table II
GPU SPECIFICATIONS

	Quadro P5000	Radeon R9 Nano	FirePro S9170
Cores	2560	4096	2816
Memory	16 GB	4 GB	32 GB
Bandwidth	288 GB/s	512 GB/s	320 GB/s
SP compute	8900 GFLOPS	8192 GFLOPS	5240 GFLOPS

hardware features of the three GPUs used, with *Bandwidth* denoting device global memory bandwidth and *SP compute* indicating theoretical single-precision peak throughput.

B. Design Modifications

In order to support additional hardware devices, we have modified the BEAGLE library at different levels (Fig. 3). At the implementation base-code layer we have changed the serial CPU solution to a *threaded model* one, using C++ threads, and throughout this paper C++ refers implicitly to the 2011 version of the standard [17]. We have also modified what was the CUDA base-code to a framework independent *accelerator model* with support for both CUDA and OpenCL external computing frameworks. This parallel implementation model communicates with the CUDA and OpenCL APIs through a single internal interface, which, in turn, has an implementation available for each framework.

Further significant sharing of code between CUDA and OpenCL exists at the kernel level. There is a single set of kernels for both frameworks, with keywords for each being defined at the pre-processor stage. Though there is a common kernel code-base for both frameworks, functions that impart a crucial effect on performance are differentiated for each hardware type. This allows for distinctly optimized parallel implementations that are shown on the figure, one for CUDA GPUs, one for OpenCL GPUs, and one for parallel x86 devices such as multicore CPUs with SIMD-extensions.

C. Library Availability

The BEAGLE project is open source under the GPL v3.0 license. The work described here will be part of an upcoming release and is available under a development branch of the library located at https://github.com/beagle-dev/beagle-lib/tree/kernel-concurrency.

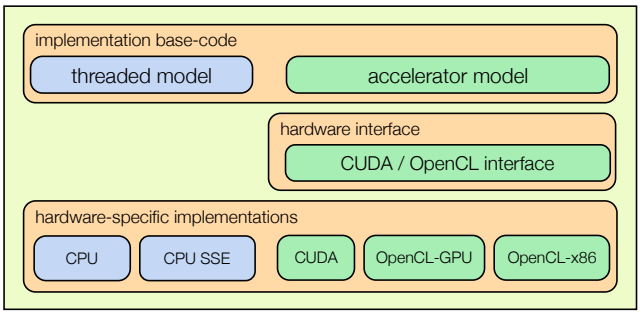

Figure 3. Layer diagram depicting the modified portions of the BEAGLE library necessary to extend hardware support.

The library includes compilation workflows for all major platforms (Linux, macOS, Windows). For Linux and macOS it uses an autoconf/automake build system. For Windows we use a Visual Studio build system. One unique aspect of the compilation system is the use of scripts to generate OpenCL/CUDA kernel source code for different inference types (e.g., amino-acid or codon-based) and floating point formats, allowing for better performance at runtime.

VI. CPU THREADED IMPLEMENTATION

To harness the increasingly parallel nature of modern CPUs, and recognizing that external frameworks such as CUDA and OpenCL are not always available to users of BEAGLE, we developed a more portable parallel implementation.

In the process of developing our solution, we briefly assessed a variety of CPU threading frameworks such as POSIX threads and OpenMP. Ultimately, we felt the best solution when balancing portability, development cost, and performance was to use the C++ threading model. This approach also allowed us to more easily combine the added parallelism with the existing, low-level, SSE vectorization of character states.

Given the decision to use C++ threads, we then iterated through a variety of approaches to concurrent computation of the phylogenetic likelihood function which we will briefly describe and compare below.

A. Futures

Our initial approach involved modifying the default CPU implementation in BEAGLE such that for each partial-likelihoods operation to be computed, a C++ standard library asynchronous future was created. Thus, this approach only concurrently computed partial-likelihood operations that were independent in the tree topology being assessed, and did not take advantage of the independent nature of each sequence pattern in the likelihood computation.

B. Thread-create

Our next approach involved the on-demand creation and joining of a set of threads with each partial-likelihoods call to BEAGLE. These C++ standard library threads were used for concurrent computation of the partial-likelihood functions across independent site patterns. We used a load-balancing approach wherein the sequence of independent patterns is broken up into equal sizes, according to the number of CPU hardware threads available. To prevent small problem sizes from being slower than the previous serial implementation, we set a minimum sequence length of 512 patterns for threading to be used.

Table III
CPU THREADING OPTIMIZATIONS

	throughput (GFLOPS)				speedup
tips	*serial*	*futures*	*thread-create*	*thread-pool*	(× serial)
8	35.82	37.92	39.07	**193.10**	5.39
16	35.47	59.70	78.26	**258.99**	7.30
64	14.95	78.67	87.91	**217.24**	14.53
128	13.62	61.61	60.19	**126.95**	9.31

C. Thread-pool

This final iteration of our CPU threading solution involved modifying the *thread-create* approach to use a pool of C++ standard library threads. For this approach we also used the threads for concurrent computation of the root likelihood across independent site patterns, in addition to the partial-likelihoods function.

Table III compares the relative performance of the core partial-likelihoods function for each of the threading approaches we assessed. The throughput measure in GFLOPS is for the single-precision floating point format and is computed as described in section V-A.

For this comparison we used a fixed sequence length of 10,000 patterns across tree sizes of 8, 16, 64, and 128 sequences at the tips, running on the two CPUs on *system 2* (Table I). The column labeled *serial* shows throughput for the original single-threaded CPU implementation in BEAGLE, with some degree of vectorization provided by GCC.

The results show the increases in performance for each iteration of our CPU threading solution and that the *thread-pool* approach performs best across all four problem sizes assessed. We also note the relative increase in performance from the original serial implementation to the final *thread-pool* solution.

VII. OpenCL Implementation

In order to exploit a broader range of hardware resources, including AMD and Intel GPUs, we extended BEAGLE so it can use the OpenCL programming framework, an open standard for parallel computing devices. With OpenCL we have also been able to better utilize the parallel computing capability of modern CPUs, both via multiple cores and vectorization extensions such as Intel SSE and Advanced Vector Extensions (AVX).

A. OpenCL and CUDA Code Sharing

The OpenCL work is based on our previous implementation for the CUDA platform. Taking advantage of the many similarities between these parallel-computing frameworks we developed a shared code design that includes a single internal interface to the hardware resource and a single set of kernels. This design allows future work on the library to more easily benefit users of either framework.

A single set of kernels for OpenCL and CUDA is achieved by using preprocessor definitions for framework specific keywords. The internal hardware interface is also shared, and only the implementation itself differs between OpenCL and CUDA. The hardware interface deals with loading the different kernels and compiling the correct one for the given analysis parameters (such as the number of states the model can assume, and floating point precision), as well as all the hardware accelerator related functions such as executing kernels, copying data, querying device characteristics, and other auxiliary functions. Few further distinctions had to be overcome for both frameworks to share code. Most notably, subpointer addressing within kernels was done by using the *clCreateSubBuffer* function in OpenCL and by pointer arithmetic in CUDA.

B. Hardware-Specific Optimizations

The use of OpenCL provides a common, vendor-neutral, platform for current and future parallel hardware architectures and allows BEAGLE to exploit a variety of resources from a single code base. Nonetheless, recognizing that important distinctions exist between what practices work best for each hardware architecture, we have adapted performance-critical code for different runtime scenarios. These hardware-specific optimizations can be categorized into two variants of our OpenCL solution, one that targets GPU architectures and another that addresses x86 processors.

1) OpenCL-GPU: With our OpenCL-GPU solution we focused on high-end NVIDIA and AMD GPUs, though our implementation is also compatible with Intel GPUs.

Our initial work on OpenCL consisted of a direct translation of the CUDA implementation running on the same NVIDIA hardware. Although we expect NVIDIA GPUs to exhibit best performance under CUDA, having them working under OpenCL served as an important comparison point and validation of our approach. The effect of framework-choice on NVIDIA devices is further explored in section VIII.

For AMD GPUs, we found that few changes were required, as these are ultimately similar in architecture to NVIDIA CUDA devices. For codon-based inference models and others with higher-count state spaces, we had to reduce the number of sequence patterns computed per work-group in our likelihood calculation kernel. This was in order to reduce memory usage in the *local* address space, as we found AMD devices to have less of this memory than NVIDIA devices.

Another optimization we implemented for AMD GPUs was the use of the OpenCL precompiler definitions FP_FAST_FMAF and FP_FAST_FMA, for single and double-precision floating-point operations respectively. These macros achieved non-trivial performance gains without loss of precision, and indicate whether fast fused-multiply-add (FMA) operations, which perform multiply and add operations in a single action, are supported by the hardware. For a problem size of 10^5 sequence patterns on a modern GPU

Table IV
OPENCL-GPU OPTIMIZATIONS

precision	patterns	throughput (GFLOPS) without FMA	throughput (GFLOPS) with FMA	% gain
single	10,000	213.02	**216.87**	1.81
double	10,000	124.14	**136.88**	10.26
single	100,000	408.63	**411.43**	0.69
double	100,000	178.04	**199.23**	11.90

Table V
OPENCL-X86 OPTIMIZATIONS

solution	work-group size (patterns)	throughput (GFLOPS)	speedup (× OpenCL-GPU)
OpenCL-GPU	64	15.75	
OpenCL-x86	64	79.65	5.06
	128	85.51	5.43
	256	**98.36**	**6.25**
	512	98.09	6.23
	1024	96.51	6.13

(AMD Radeon R9 Nano on *system 1*), we noticed up to an 11.9% performance improvement in double-precision mode for our core partial-likelihoods kernel (Table IV).

2) OpenCL-x86: For our OpenCL-x86 solution we collaborated with Intel to develop an optimized implementation for Xeon CPUs and first generation Xeon Phi (Knights Corner) accelerators. Since this work started Intel has dropped support for OpenCL on Xeon Phi, however Intel has continued developing drivers for Xeon CPUs and we have observed strong performance on these multicore processors with our x86 solution (further detailed in section VIII).

In the process of finding the best solution for Intel x86 processors, we tested several variations of our core partial-likelihoods kernel. This included explicit OpenCL vector usage and reorganization of execution threads from two to three-dimensional work-groups. Ultimately we found that the key optimization was to have each thread of execution do more work in comparison to our GPU approach. This was especially important when computing the partial-likelihoods function for nucleotide models where only 4 states are possible and each thread has a lighter workload. To achieve this heavier workload per thread, our OpenCL-x86 for DNA-based inferences, loops over the state space in each work-item instead of computing all states concurrently, as is done with the GPU approach. We also found that it was advantageous to avoid the explicit use of the *local* memory address space and allow the OpenCL compiler to manage memory caching.

Given these x86-specific changes to our nucleotide-model likelihood computation kernel, we proceeded to optimize for work-group size, which determines the number of sequence patterns computed per work-group. Table V explores performance with the dual CPUs on *system 2* for work-groups of increasing size. The table also shows throughput for our original OpenCL-GPU solution running on the Xeon CPUs and the relative speedup achieved due to our architecture-specific optimizations. We observe that peak performance is achieved with a work-group size of at least 256 patterns. We opted to use this size as we prefer the smallest work-group size with peak or near-peak performance to reduce pattern padding when the total number of patterns is not divisible by the work-group size.

3) OpenCL Driver Implementations: BEAGLE makes use of the OpenCL Installable Client Driver loader to make all implementations on a system available, which allows the selection of different drivers for the same hardware resource.

On Linux and Windows operating systems we have found that vendor-specific OpenCL driver implementations offer the best performance. On macOS vendor-specific drivers are not available and we observed reduced performance compared to other platforms.

VIII. RESULTS

Here we explore the performance of the new implementations for the BEAGLE library on a variety of modern parallel hardware resources. System specifications are as described in Table I. We also evaluated performance on a machine with an Intel Xeon Phi 7210 CPU (not an accelerator), Linux kernel version 3.10.0, and GCC version 6.2.0.

A. Partial-likelihoods Kernel Performance

We have used our *genomictest* program to benchmark the core likelihood function of BEAGLE on a variety of hardware platforms and for a range of problem sizes. Again, this function is the main bottleneck for phylogenetic inferences, typically accounting for over 90% of the total execution time. We have found the relative performance gains observed here correlate strongly with those of a full inference run.

Figure 4 shows throughput in effective GFLOPS (billions of floating-point operations per second) for our partial-likelihoods calculation kernel for analyses with increasing unique site pattern counts, across a number of parallel computing devices and implementations. We evaluated our C++ threading, OpenCL-x86, OpenCL-GPU, and CUDA implementations. The hardware devices represent a sample of the range of consumer-level and high-performance computing resources available to domain scientists who are the ultimate users of the BEAGLE library, and included AMD Radeon R9 Nano, AMD FirePro S9170, and NVIDIA Quadro P5000 GPUs, Intel Xeon Phi 7210 manycore CPU, and dual Intel Xeon E5-2680v4 multicore CPUs. The figure includes performance results for computing partial-likelihoods for both nucleotide and codon-model analyses. The left-side vertical axis labels show the speedup relative to the average performance of a baseline serial, single threaded and non-vectorized, CPU implementation. We chose to use this non-parallel CPU implementation as a comparison baseline as

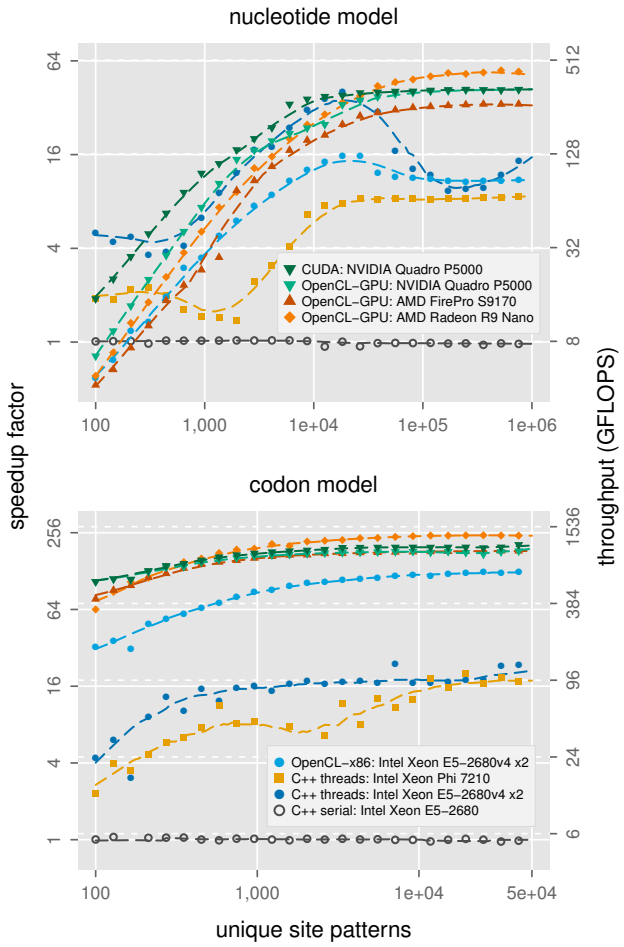

Figure 4. Plots showing throughput performance in GFLOPS for the core likelihood function of the BEAGLE library, for nucleotide and codon-based models with a range of problem sizes running on a variety hardware platforms and implementations. Speedup factors (which are relative to unvectorized single-core performance), throughput, and number of unique site patterns are on a log-scale.

it provides a consistent performance level across different problem sizes. It is also relevant as it has been the default implementation in BEAGLE in previous releases and many phylogenetic inference softwares use serial code as their standard. We note that performance results reported here are in all cases far from theoretical peak compute throughput for each platform as the calculation of phylogenetic sequence likelihoods is substantially memory-bound, especially for nucleotide models.

1) Nucleotide Model: For nucleotide-based likelihood, we observe that throughput strongly scales with the number of site patterns for all parallel hardware resources using our *accelerator model*. For a small number of patterns the parallel OpenCL implementations exhibit poor performance relative to others due to greater execution overhead. By 10^5 patterns the performance across these devices has reached a saturation point, with the exception of the AMD Radeon R9 Nano GPU, which continues to slightly scale up in performance. Overall, best performance is achieved by the AMD Radeon R9 Nano GPU, with 444.92 GFLOPS of throughput for a problem with 475,081 unique site patterns. This represents a ∼58-fold speedup over the baseline serial, non-vectorized, CPU implementation, and a ∼5.1-fold speedup over our fastest CPU solution at this problem size, which is the OpenCL-x86 implementation running on two Intel Xeon E5-2680v4 processors.

For CPUs using our *threaded model*, we observe that performance does not monotonically increase with the number of patterns and we require further investigation to understand this aspect of the result. We observe very strong performance for the dual Intel Xeon E5-2680v4 CPUs between approximately 3,000 and 50,000 patterns, with a peak performance of 328.78 GFLOPS at 20,092 unique patterns, also being the overall fastest implementation at this problem size. We observe weak performance from the Xeon Phi 7210 CPU for problems under 10^4 patterns, though we have not done optimization work specific to this platform. Further, we note that we did not use SSE vectorization for these benchmarks as it is not available in single-precision in BEAGLE.

2) Codon Model: For codon-based analyses, we observe that throughput performance is less sensitive to the number of unique site patterns. This is due to the better parallelization opportunity afforded by the 61 biologically-meaningful states that can be encoded by a codon. This higher state count of codon data compared to nucleotide data increases the ratio of computation to data transfer resulting in increased relative performance for codon-based analyses (Fig. 4). We also observe similar performance from all GPU devices and less overhead effect from use of the OpenCL framework. Overall, highest throughput is achieved by the AMD Radeon R9 Nano GPU, with 1324.19 GFLOPS for 28,419 patterns, equivalent to a ∼253-fold speedup over the baseline serial, non-vectorized, CPU implementation and ∼2-fold speedup over the OpenCL-x86 implementation running on two Intel Xeon E5-2680v4 processors. Our *threaded model* for CPUs does not perform as well for codon-based inferences as it only parallelizes the computation of independent site patterns.

B. Multicore Performance Scaling

Figure 5 shows CPU performance results of the core likelihood function for nucleotide-model analyses with 10^4 unique patterns when utilizing an increasing number of hardware threads on *system 2*. The two processors on this system have 14-cores each for a total of 56 hardware threads running at 2.40 GHz. This benchmark was achieved using the *taskset* utility in Linux for our *threaded model* implementation, and the OpenCL device-fission feature for our OpenCL-x86 solution.

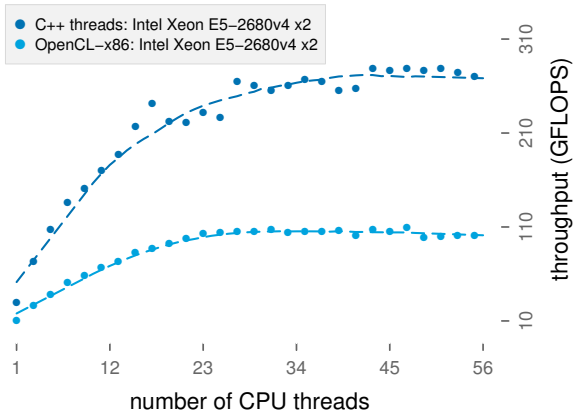

Figure 5. Plot showing multicore CPU performance scaling for the *threaded model* and OpenCL-x86 implementations in BEAGLE for nucleotide-model likelihood function with 10^4 patterns. Throughput in GFLOPS is on a log-scale.

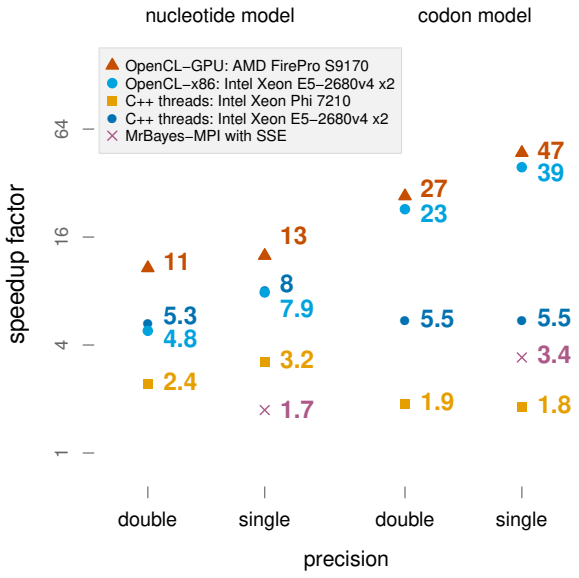

Figure 6. Plot showing speedups for MrBayes in single and double-precision mode using various BEAGLE library implementations as well as the built-in SSE option, relative to the MrBayes-MPI implementation in double-precision. Speedup factors are on a log-scale.

Parallelization on multicore systems remains an important topic as researchers increasingly invest in multicore hardware, where core counts on high-end systems regularly reach 40 or greater. The results here show that throughput for both implementations starts to saturate at around 27 threads, suggesting memory bandwidth limitations.

C. Application-Level Results

We ran MrBayes 3.2.6 on *system 2* to benchmark application-level performance for our new C++ threaded, OpenCL-x86, and OpenCL-GPU implementations for BEAGLE. MrBayes uses MPI to concurrently compute separate Markov chain Monte Carlo chains across processors [18]. This is an additional level of concurrency and is complimentary to that provided by the BEAGLE library, which parallelizes computation across site patterns with our *threaded model* implementation, and across site patterns, states, and rate categories with our *accelerator model* solutions. Additionally, MrBayes uses SSE vectorization in single-precision floating point format.

For evaluating performance with the nucleotide model we used a dataset from an RNA-Seq study of advanced moths and butterflies [19] with 16 taxa and 742,668 site patterns, of which 306,780 were unique. For the codon model benchmark we used a 15 taxa dataset with 6,080 unique codon patterns, which was a subset of a larger arthropod dataset [20]. Both analyses were run with four Metropolis-coupled, Markov chain Monte Carlo chains.

We also assessed each dataset under single and double-precision floating point formats. MrBayes supports both modes and certain analyses with larger number of taxa benefit from more precise computation. All reported speedups compare the total execution time relative to that of MrBayes-MPI in double-precision mode.

Generally we observe that speedups are largest under the codon models, as they allow for greater parallelism. For the OpenCL-GPU implementation we note significant speedups across all benchmark scenarios. Relative to the fastest single-precision format implementation in MrBayes, speedups are 7.6 and 13.8-fold for the nucleotide and codon model analyses, respectively. For the CPU-based implementations, we observe that for a nucleotide analysis of this problem size both implementations are closely matched, although for codon inferences, the OpenCL-x86 has a significant advantage. We also observe relatively modest performance from the Xeon Phi CPU across all scenarios.

IX. Conclusion

The BEAGLE project addresses a common bottleneck across phylogenetic inference programs by accelerating likelihood computation. The library now includes additional parallel computing implementations, and combines both CUDA and OpenCL frameworks in a single codebase to address a wider-range of hardware resources. These advancements are of immediate benefit to users of phylogenetic programs that exploit the library. Additionally, developers of other phylogenetic software packages can reference these results to assess the suitability of using BEAGLE with their program, or for developing similar parallel solutions.

Although the improvements described in this paper also allow users to execute in parallel on multiple devices within a system, this requires the client program to partition the problem across site patterns and create a separate library instance for each hardware device. Further, selecting the

best performing implementation depends not only on the hardware available but on problem size and type. We plan to further develop BEAGLE so that computation can be dynamically load balanced across multiple devices from within a single library instance. The library would also select the best implementation for each data subset and hardware pair. This will allow for greater memory efficiency and performance gains which will be especially relevant in heterogeneous systems.

ACKNOWLEDGMENT

We thank Marc Suchard, University of California, Los Angeles, and Andrew Rambaut, University of Edinburgh; Yariv Aridor and Arik Narkis, Intel Israel, for assistance with Intel Xeon Phi programming; Mark Berger of NVIDIA; and Greg Stoner and Ben Sander of AMD. This work was supported by the National Science Foundation grant numbers DBI-0755048 and DBI-1356562.

REFERENCES

[1] 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," *Syst. Biol.*, vol. 61, pp. 170–173, 2012.

[2] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 353–364.

[3] R. Domínguez, D. Schaa, and D. Kaeli, "Caracal: Dynamic translation of runtime environments for gpus," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 5:1–5:7.

[4] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu, "Languages and compilers for parallel computing," J. N. Amaral, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, ch. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pp. 16–30.

[5] G. Martinez, M. Gardner, and W. C. Feng, "CU2CL: A CUDA-to-OpenCL translator for multi-and many-core architectures," in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 2011, pp. 300–307.

[6] M. Gardner, P. Sathre, W. C. Feng, and G. Martinez, "Characterizing the challenges and evaluating the efficacy of a CUDA-to-OpenCL translator," *Parallel Computing*, vol. 39, no. 12, pp. 769–786, 2013.

[7] M. Harvey and G. D. Fabritiis, "Swan: A tool for porting CUDA programs to OpenCL," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093 – 1099, 2011.

[8] J. Felsenstein, "Evolutionary trees from DNA sequences: a maximum likelihood approach." *J. Mol. Evol.*, vol. 17, no. 6, pp. 368–76, 1981.

[9] D. J. Zwickl, "Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion," Ph.D. dissertation, University of Texas, Austin (TX), 2006.

[10] A. J. Drummond, M. A. Suchard, D. Xie, and A. Rambaut, "Bayesian phylogenetics with BEAUti and the BEAST 1.7," *Mol. Biol. Evol.*, vol. 29, pp. 1969–1973, 2012.

[11] F. Ronquist, M. Teslenko, P. van der Mark, D. Ayres, A. Darling, S. Höhna, B. Larget, L. Liu, M. Suchard, and J. Huelsenbeck, "MrBayes 3.2: efficient Bayesian phylogenetic inference and model choice across a large model space," *Syst. Biol.*, vol. 61, no. 3, pp. 539–542, 2012.

[12] S. Guindon, J.-F. 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," *Syst. Biol.*, vol. 59, no. 3, pp. 307–321, 2010.

[13] D. L. Ayres and M. P. Cummings, "Configuring concurrent computation of phylogenetic partial likelihoods: Accelerating analyses using the BEAGLE library," in *2017 17th International Conference on Algorithms and Architectures for Parallel Processing: ICA3PP Collocated Workshops*, Helsinki, Finland, in press.

[14] J. Bao, H. Xia, J. Zhou, X. Liu, and G. Wang, "Efficient implementation of MrBayes on multi-GPU," *Molecular Biology and Evolution*, vol. 30, no. 6, p. 1471, 2013.

[15] L. Kuan, F. Pratas, L. Sousa, and P. Tomás, "MrBayes sMC3: Accelerating Bayesian inference of phylogenetic trees," *The International Journal of High Performance Computing Applications*, 2016.

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

[17] C++ Standards Committee and others, "ISO International Standard ISO/IEC 14882: 2011, Programming Language C++," Geneva, Switzerland, Tech. Rep., 2011.

[18] G. Altekar, S. Dwarkadas, J. P. Huelsenbeck, and F. Ronquist, "Parallel Metropolis coupled Markov chain Monte Carlo for Bayesian phylogenetic inference," *Bioinformatics*, vol. 20, no. 3, pp. 407–415, 2004.

[19] A. L. Bazinet, M. P. Cummings, K. T. Mitter, and C. W. Mitter, "Can RNA-Seq resolve the rapid radiation of advanced moths and butterflies (Hexapoda: Lepidoptera: Apoditrysia)? An exploratory study," *PLoS ONE*, vol. 8, no. 12, p. e82615, Dec 2013.

[20] J. Regier, J. Shultz, A. Zwick, A. Hussey, B. Ball, R. Wetzer, J. Martin, and C. Cunningham, "Arthropod relationships revealed by phylogenomic analysis of nuclear protein-coding sequences," *Nature*, vol. 463, pp. 1079–1083, Feb 2010.